conference

proceedings

4th USENIX Windows Systems Symposium

Seattle, Washington, USA
August 3–4, 2000

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX 4th USENIX Windows Systems Symposium Seattle, Washington, USA August 2000

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $18 for members and $24 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past Windows Systems Proceedings

3rd USENIX Windows NT Symposium	1999	Seattle, Washington, USA	$18/24
2nd USENIX Windows NT Symposium	1998	Seattle, Washington, USA	$18/24
USENIX Windows NT Workshop	1997	Seattle, Washington, USA	$18/24

ISBN 1-880446-20-0

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

4th USENIX

Windows Systems Symposium

August 3–4, 2000
Seattle, Washington, USA

Conference Organizers

Program Chairs

J. Bradley Chen, *Appliant.com*
Richard Draves, *Microsoft Research*

Program Committee

Ed Felten, *Princeton University*
Jim Gray, *Microsoft Research*
Brad Myers, *Carnegie Mellon University*
Karin Petersen, *Xerox Palo Alto Research Center*
Mendel Rosenblum, *Stanford University*
Dave Steere, *Oregon Graduate Institute*
Werner Vogels, *Cornell University*
Bill Weihl, *Akamai Technologies, Inc.*
Rumi Zahir, *Intel Corporation*

Steering Committee

Andrew Hume, *AT&T Labs Research*
Michael B. Jones, *Microsoft Research*
Werner Vogels, *Cornell University*

The USENIX Association Staff

4th USENIX Windows Systems Symposium

August 3–4, 2000
Seattle, Washington, USA

Message from the Symposium Chairs

Welcome to Seattle and the 4th USENIX Windows Systems Symposium.

Since 1997 this symposium has provided a unique forum for our community to share interests in Windows research and advanced development. In this fourth year, we have decided to preserve the focus on the Win32 platform while expanding the scope of the symposium beyond Windows NT to include all Win32-based systems—Windows 98 and Windows CE as well as Windows NT and Windows 2000. The result is a program that covers the spectrum from tiny devices embedded in a "smart" home to the large cluster-based systems required to handle Internet service delivery.

This year the program committee reviewed twenty-five full paper submissions. Each paper was reviewed by at least four members of the program committee. We met in March and selected twelve papers for the program, each with a shepherd assigned from the program committee. Five of the accepted papers came from industry and the remainder from academia. We would like to thank the authors for their drive and creativity and the program committee and external reviewers for their diligent and conscientious efforts.

In addition to the research papers, we are pleased to include three very relevant invited talks in our program this year. We welcome Mark Lucovsky, Tilak Mandadi, and Andrew Chien and thank them for taking this opportunity to share their outstanding work with our community. To round out the program, Werner Vogels has planned the work-in-progress and poster sessions.

We would like to acknowledge the efforts of key individuals who supported us in this project. Jonathan Simon provided much timely assistance with the Web database system we used for tracking submissions and reviews. The USENIX staff, with Judy DesHarnais, Ellie Young, Monica Ortiz, Jane-Ellen Long, Toni Veglia, Jennifer Radtke, and Vanessa Fonseca, maintained its reputation for outstanding support. Todd Wanke helped us find our invited speakers. Mike Jones drew from his depth of experience to provide advice at key moments in our preparations.

Lastly we'd like to thank our employers, Appliant.com and Microsoft, for recognizing the value of academic and industrial research and supporting our efforts to bring you this conference.

Please enjoy Seattle and have a great symposium!

J. Bradley Chen, Appliant.com
Richard Draves, Microsoft Research
Symposium Co-Chairs

Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services

Minwen Ji, Edward W. Felten, Randolph Wang, and Jaswinder Pal Singh
Department of Computer Science, Princeton University
{mji, felten, rywang, jps}@cs.princeton.edu

Abstract

Maintaining availability in the face of failures is a critical requirement for Internet services. Existing approaches in cluster-based data storage rely on redundancy to survive a small number of failures, but the system becomes entirely unavailable if more failures occur. We describe an approach that allows a cluster file server to *isolate* failures so that the system can continue to serve most clients. Our approach is complementary to existing redundancy-based methods: redundancy can mask the first few failures, and failure isolation can take over and maintain availability for the majority of clients if more failures occur.

The building blocks of our design are self-contained and load-balanced file servers called *islands*. The main idea underlying island-based design is the *one-island principle*: as many operations as possible should involve exactly one island. The one-island principle provides failure isolation because each island can function independently of other islands' failures. It also helps the file system scale with the system and workload sizes because communication and synchronization across islands are reduced. We implemented a prototype island-based file system called *Archipelago* on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The measurement of micro benchmark shows that Archipelago adds little overhead to NTFS and Win32 RPC performance; while the measurement of operation mixes based on NTFS traces shows a speedup of 15.7 on 16 islands.

1. Introduction

NT clusters are an important tool for large I/O-intensive applications such as file servers, Web servers, and other Internet services. A wide variety of research projects on cluster file systems have explored approaches to building cluster file systems that provide high availability and scalability.

This paper discusses a new approach to maximizing availability on a cluster file server. We use the percentage of requests that succeed despite the failure of one or more servers as the availability metric, our goal in this work is maximize this percentage.

There are two complementary approaches to maximizing availability. First, we can use redundancy to maintain complete availability in the face of a small number of failures; second, we can try to *isolate* failures in order to serve as many requests as possible even though some cannot be served. These approaches are complementary, since we can use redundancy to mask the first few failures, and then use isolation to cope with any additional failures.

This paper describes an approach to cluster file system design that provides failure isolation. We divide the nodes in the system into groups called *islands*. An island might be a single node, or it might be a group of nodes that use redundancy within the island to mask failures. In either case, island-based design strives to serve as many client requests as possible when one or more islands have crashed or are unavailable.

The main idea underlying island-based design is the *one-island principle*: as many file system operations as possible should require the participation of exactly one island. The one-island principle provides good failure isolation because each island can function independently of other islands' failures. In other words, the failure of 1 out of n islands in an island-based file system renders only 1/n data inaccessible. The one-island principle allows island-based systems to scale efficiently with the system and workload sizes because communication and synchronization across islands are reduced.

Our motivation of failure isolation is analogous to the motivation of fault containment in Hive [26]. Hive, an operating system for large-scale shared-memory multiprocessors, attempts to "contain" a failed part so that it does not bring down other parts.

The target application of an island-based file

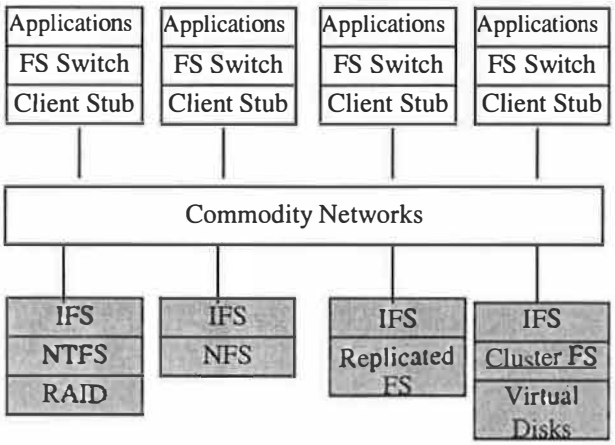


Figure 1. Overview of an island-based file system (IFS). Shaded boxes are islands or servers and non-shaded boxes are clients.

system is the data storage for those Internet services that prefer to serve as many clients as possible rather than to go entirely offline when partial failures are present, that are medium to large scale, e.g. tens to hundreds of PC's connected by commodity local area networks such as Ethernet, and that expect occasional node failures and network partitions. Examples include email, Usenet newsgroup, e-commerce, web caching, and so on.

We evaluated the island-based design by statistical analysis of the access patterns of existing systems. The results show that the partial availability provided by island-based file system is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients. In one example, if 1 out of 32 islands is down for an hour, we expect that 93.8% clients during that hour will not notice the temporary partial failure. On average 99.8% operations involve a single island and hence do not require communication or synchronization across islands.

We implemented a prototype of island-based file system called *Archipelago* on a cluster of PCs running Windows NT 4.0 connected by Ethernet. The measurement of micro benchmarks shows that Archipelago adds little overhead to NTFS and Win32 RPC performance; the measurement of operation mixes based on NTFS traces shows a speedup of 15.7 on 16 islands.

2. Quantified motivation

To quantitatively motivate the potential advantage of island-based design, let us examine the temporary or permanent data loss ratios under partial failures in existing cluster file systems and island-based file systems with the same redundancy schemes.

We modeled the data loss ratio in case of partial failures in cluster file systems (CFS) built on top of virtual storage layers, such as Frangipani [1] and xFS [4], under various redundancy schemes. The results show that CFS loses a significantly larger portion of data than the virtual storage it loses in a partial failure because the data in a surviving server will be inaccessible if any server containing a piece of metadata needed to access the surviving data fails. For example, with the loss of 1 out of 32 non-redundant virtual storage servers or 3.1% non-redundant virtual storage space, CFS is expected to lose 63.8% data in files and directories. The detailed analytic models can be found in our technical report [14].

We suggest that the temporary or permanent data loss of an existing redundant file system can be reduced by a failure isolation scheme without altering the underlying redundancy scheme. We observe that many existing redundant storage systems are divided into groups and that data redundancy is applied within groups, but not across groups. It results either from the nature of the redundancy scheme, such as mirroring pairs, or from performance optimization, such as RAID-5 striping groups [4]. By configuring each redundant group as an independent file system, we can always achieve better availability than by running a single file system on top of the whole storage system. We change the data loss example above by assuming that each of the 32 virtual storage servers is a RAID-5 single-parity stripe group of 4 physical nodes and that xFS [4] runs on the 128 nodes as a single file system. If we run an independent xFS in each group, we expect to lose only 3.1% vs. 63.8% data when we lose at least 2 nodes in the same group.

The challenge is how to evenly, automatically and dynamically partition a single large file system into a cluster of independent components without causing inconsistency across components in the face of partial failures.

3. System structure

Figure 1 gives an overview of an island-based file system in a typical configuration. An island consists of a server process running on top of a local file system. Client applications view the island-based file system as a single system and access it through local file system switches and

stubs. Islands and clients are connected by commodity local area networks such as Ethernet.

Let us examine two important issues in island-based design, data distribution and metadata replication.

3.1 Hash-based data distribution

We designed a new data distribution strategy for island-based file systems: data is distributed to islands at *directory granularity* by *hashing* the *pathnames* of the directories to island indices.

We choose directory granularity rather than block, file or sub tree granularity because most file system operations involve a single directory and hence satisfy the one-island principle, and directories are finer grained than sub trees so as to allow load balance.

We choose hashing instead of recursive name lookup because hash functions can be computed on the client machines without contacting any servers. We choose to hash pathnames instead of low-level integer identifiers such as inode numbers because pathnames are the only information that a client can possibly have without contacting any servers, and they are independent of internal representations of file systems.

Clients determine which island to contact for a directory or a file in that directory by hashing the full pathname of the directory to an island index in two steps: first, hashing the pathname to a *bucket* (an integer) with a universal hash function [7]; second, hashing the bucket to an island index with an extendible hash table [8]. The universal hash function used in an island-based file system is a consistent mapping from a variable-length character string to a 32-bit integer and has good distribution in the output space independently of the input space. A universal hash function can evenly distribute an arbitrary set of directories to buckets; however, it does not have control on the workload distribution across directories; therefore, an additional level of indirection is necessary to handle the hot spots and dynamic load changes. A subset of the 32 bits is used as the index to the extendible hash table and the table entries are island indices. As load imbalance across islands increases or islands are permanently added or removed during system reconfiguration, the table entries are reassigned to islands to rebalance the load using a bin-packing algorithm. The reassignment is made *monotonic*, i.e. each island either loses data or gains data, but not both.

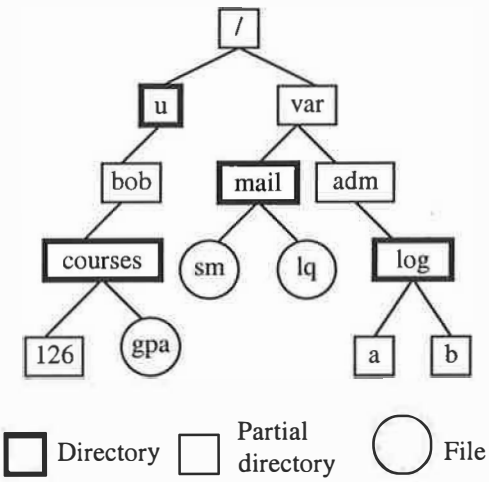


Figure 2. Skeleton hierarchy and directory replication. This is an image of the internal file system in an island that is the directory owner of the highlighted directories. Partial directories are replicas that contain only attributes and partial contents or no contents.

Therefore, only a minimal amount of data needs to be migrated between islands.

Inside each island, we store directories in a *skeleton hierarchy*. We call the file system running inside each island the *internal file system*. An internal file system can be an instance of any existing file system such as a local file system, a replicated file system or a cluster file system. The skeleton hierarchy in an island contains the directories hashed to this island index and their ancestor directories up to the root, and is stored in the unmodified internal file system as a normal tree. This way, islands can function independently of others' failures and we can leverage the functions of the internal file systems. The consequence of storing data in skeleton hierarchies is the replication of certain metadata or directory attributes.

3.2 Usage-based metadata replication

Although it might not take much space to replicate metadata across islands if it accounts for a small portion of the entire system, updates to replicated metadata will have to be done in all replicas and hence violate one-island principle. Therefore, we use a *usage-based* adaptive replication scheme in the island-based design, i.e. we replicate metadata that is more frequently used to a higher degree.

To help us explain the usage-based metadata replication, we introduce two terms, *directory*

owner and *parent owner*. The *directory owner* of a directory is the island to which the directory is hashed. The *parent owner* of a file or directory is the directory owner of its parent directory. A file resides in exactly one island, its parent owner. A directory will be replicated in its parent owner, in its directory owner and in all the parent owners of its descendent directories. Therefore, the replication scheme can automatically adapt to the usage of the metadata. In particular, the root directory is replicated in all islands; files are not replicated across islands; intermediate directories are replicated to various degrees.

However, only some directory *attributes*, not the directory *contents*, need to be replicated. Directory contents are the lists of names and addresses of sub directories and files. Only the directory owner keeps a complete copy of the directory contents; other replicas have partial contents or no contents. Changes to directory contents, e.g. adding or removing files, need to be done in the directory owner only. Directory attributes include name, size, security, time stamps, read-only tag, compressed tag, etc.. Changes to directory attributes will, however, affect multiple replicas.

We want to replicate only those attributes that are needed when a descendent of the directory is looked up. We divide directory attributes into two categories, *static* attributes and *dynamic* attributes, based on their access patterns. A static attribute is more frequently read than written, and a dynamic attribute is more frequently written than read. Attributes such as name, security, read-only tag and compressed tag are static. Attributes such as size and time stamps are dynamic. We replicate the static attributes and do not replicate the dynamic attributes. We use a *read-one-write-all* policy to maintain consistency of the static attributes; the overhead of updates is acceptable since static attributes rarely change. We read and write dynamic attributes in a single island, the directory owner.

Figure 2 gives an illustration of the skeleton hierarchy and metadata replication.

3.3 Evaluation

We evaluated the load balance and storage overhead in island-based file systems by statistical analysis of the contents of existing systems. Detailed measurements and analysis can be found in our technical report [14]. We summarize the results as follows:

- Only a small portion of storage is needed for replicating directory attributes (0.1% to 0.5% per island or 0.3% to 7.7% in total in our experiments).
- Load imbalance (average number of bytes per island dividing its standard deviation) resulted from the hashing algorithm in island-based file systems is low (0.0001 to 0.0279 in our experiments) in spite of the unbalanced load across directories or hot spots.

4. Protocols and other design issues

To make the island-based design a viable solution, we need to address the issues of rebalance, consistency, recovery, etc. in addition to data distribution and metadata replication. We use standard approaches that are tailored to island-based file systems, as we will briefly describe below.

4.1 Rebalance protocol

As discussed in the previous section, the hash function in data distribution can be changed to rebalance the load across islands when load imbalance exceeds a threshold or when islands are permanently added to or removed during system reconfiguration.

We use a two-phase commit protocol [16] in the rebalance procedure so that the hash table is updated in all islands atomically in the face of partial failures. In the first phase, load information (number of bytes) is collected from all islands and all islands are prepared for the rebalance. In the second phase, a new hash table is computed according to the load information, and is either updated in all islands or aborted if any island is inaccessible.

The hash table is replicated in all clients of the file system as well as in all islands. The table has an entry per directory bucket and the number of buckets is a constant factor of the number of islands; therefore, the table size is proportional to the number of islands. (The universal hash function can map multiple directories to the same bucket. See Section 3.1.) A client is asked to update its hash table when any server detects its out-of-date copy using piggy-back information in regular operations.

How often the rebalance procedure needs to be invoked depends on the load imbalance that can be tolerated. We expect that a reasonable threshold can be set so that the rebalance procedure is invoked at a non-disruptive frequency, e.g. once

every weekend.

A trace-driven study of the online reconfiguration of a web server running on an island-based file system shows that data migration in the rebalance procedure is made transparent to the web server in terms of both functionality and performance [14]. Therefore, we do not expect the rebalance procedure to have a noticeable impact on client operations.

4.2 Consistency protocol

Since certain states, e.g. static directory attributes, are replicated across islands, a cross-island protocol is necessary to keep the replicas consistent in the face of island failures and network partitions. Cross-island operations in island-based file systems include *CreateDir* and *RemoveDir*, which involve two islands, *SetDirAttr*, *SymLinkDir* and *DeleteLinkDir*, which involve all islands, and *RenameDir*, which involves a variable number of islands depending on the directory to be renamed.

The island-based design eases the consistency maintenance in two ways. First, the majority of operations involve a single island, hence do not require a cross-island protocol for consistency. Second, all cross-island operations on the same object are coordinated by a single island, i.e. the directory or parent owner; hence synchronization can be done with centralized control per object, which eases the protocol design.

The single coordinator property of the protocol ensures that no conflicting updates will occur even in the face of network partitions, hence largely relaxes the synchronization semantics. We designed and implemented a protocol that uses logical clock synchronization [15], logging [10] and two-phase commit [16] for atomicity and serialization of cross-island operations. In particular, we choose to maintain the following invariants in the face of island failures and network partitions:

1. All operations on the same object are serialized, i.e. clients observe them in the same order in all islands.
2. All operations by the same client thread are serialized, i.e. clients observe them in the same order in all islands.
3. Operations by different clients can be serialized if the clients interact with each other by accessing the same object(s) in the file system.
4. The ordering relations are *transitive*, i.e. if operation 1 is observed to happen before 2 and 2 before 3 then 1 is observed to happen before 3.

4.3 Recovery protocol

We designed and implemented a fairly standard recovery protocol for islands to recover from various combinations of failures back to consistent state.

Cross-island operations are logged on disk if they cannot be committed in all involved islands due to island failures or network partitions. A failed or disconnected island will exchange logs with other islands upon reconnection to those islands. In particular, we choose to maintain the following invariants in the state transitions of a recovering island *r*:

1. All logged operations from other islands will be committed in *r* in the ascending order of their time stamps. That is, operations serialized in real time will be committed in the same order as if *r* had not failed.
2. No client requests or requests that indirectly affect clients' view of the system state will be processed in *r* until all logged operations have been committed in *r*. That is, the inconsistent state of *r*, if there is any, is made invisible to clients.

4.4 Other design issues

Island-based file systems inherit most functions from their internal file systems, such as metadata structures, disk space allocation, I/O scheduling, server-side caching, locking, local security, recovery, etc.; therefore, we are not concerned about all the low-level details in file system design and implementation. We extended certain functions, such as symbolic links and renaming directories, to adapt to the island-based environment. Interested readers should refer to our technical report for more information about the design, implementation, and evaluation of our prototype [14].

5. Implementation

We have implemented a prototype of island-based file system called *Archipelago* on a cluster of Pentium II PCs running Windows NT 4.0. NTFS [13] is used as the internal file system. NTFS uses extensive caching and name indexing for better performance and logs metadata changes for local recoverability. NTFS can be configured to run on a group of disks with parity striping for data redundancy.

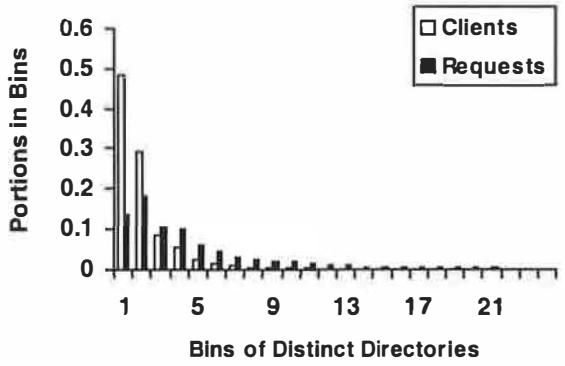


Figure 3. Histograms of clients and requests by bins of distinct directories in the web traces. The numbers read as "48.3% clients accessed 1 distinct directory during every hour" or "17.9% requests were issued by clients who accessed 2 distinct directories during every hour". Accesses to more than 24 directories account for 0.4% clients and 19.3% requests in total, and are omitted in the graph for readability.

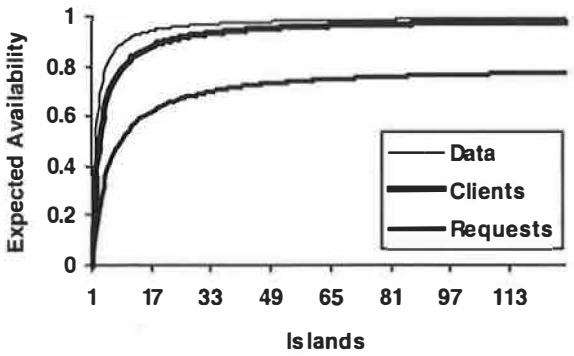


Figure 4. Expected availability for data, clients and requests in the web traces with the failure of 1 out of *n* islands. The x axis is the number *n* of islands. The y axis is the expected availability, i.e. (1-1/*n*) for data and $\sum p(i)\cdot(1-1/n)^i$ for clients and requests, where *i* is the bin of distinct directories and *p(i)* is the portion of clients or requests in the bin *i*.

An Archipelago server runs on each machine and forms an island. Each client accesses files through a local stub, which forwards the request to a server through Windows remote procedure call (Win32 RPC). The server is implemented as a user-level process. For expediency, our prototype client is implemented as a stub .dll that redirects requests for Archipelago files directly to servers, bypassing the in-kernel file system drivers. This solution is adequate for experimental purposes, although it does not provide total seamless integration with existing applications. A more complete solution would implement a full installable file system driver [20]. We believe the performance difference in these two solutions to be negligible compared with the time to service file system requests in a distributed file system.

The server and stub are implemented in C++, and consist of 3088 and 5415 lines of code, respectively. The server program is linked with the stub library for code reuse purpose. In addition, there are 24042 lines of automatically generated C code for RPC and system call interception.

6. Measurements

In this section, we present the selected measurements to answer the following questions.

1. How many clients will likely notice a partial failure in an island-based system? (Section 6.1)
2. What is the overhead of island-based design in simple cases? (Section 6.2)
3. How many operations require cross-island communication and synchronization? (Section 6.3)
4. How do cross-island operations affect the overall scalability of an island-based file system? (Section 6.4)

6.1 Impact of partial availability on web clients

The effective availability of an island-based file system with partial failures depends on the number of distinct directories that clients access because a partial failure in the system causes a random set of directories to be inaccessible.

We compute the histograms of clients and requests by the distinct directories they touched from the access logs of the web server running on our site [23]. We assume that the island-based file system acts only as a content provider to the web server, i.e. accesses to control information or executables of the web server itself do not count in our statistics. We group the HTTP requests into clients by the hostnames or IP addresses in the requests, and within each client, we group requests into directories by the URLs in the requests. We compute the histograms from two months' traces, July 1998 (137248 clients and 1304975 requests in total) and January 1999 (166804 clients and 1297428 requests in total), using a time window size of an hour. The results, in Figure 3, show that the largest portion (48.3%) of clients accessed only

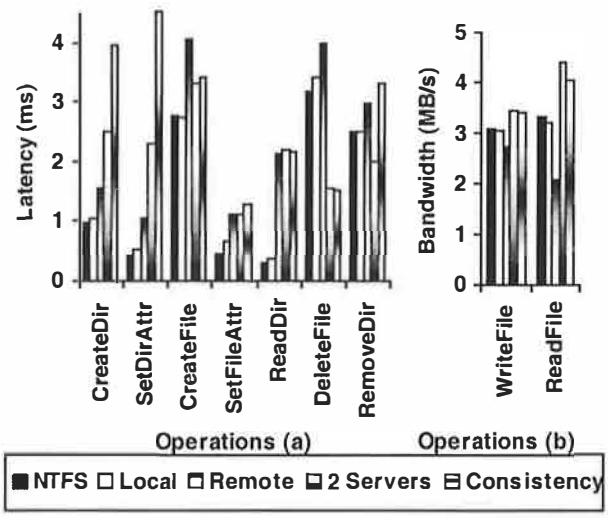


Figure 5. Single client performance. A single client runs the micro benchmarks in five cases: directly on NTFS (NTFS), on the local machine of an Archipelago server (Local), on a remote machine from the server (Remote), with two servers (2 Servers), and with the consistency protocol turned on with two servers (Consistency), respectively. The y-axis in (a) is the latency in milliseconds measured at the client side. Lower columns represent better performance. The y-axis in (b) is the bandwidth in MB/s in the WriteFile and ReadFile operations measured at the client side. Higher columns represent better performance.

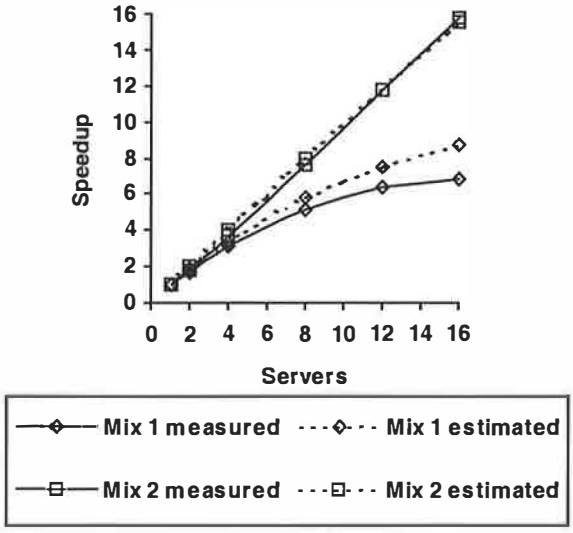


Figure 6. Speedup of throughputs of randomized operation mixes. The four curves are the measured speedup of operation mix 1 (Table 2), estimated speedup of operation mix 1, measured speedup of operation mix 2 (Table 2), and estimated speedup of operation mix 2, respectively. The speedup is calculated as the absolute throughput (requests/sec) divided by the throughput of 1 server. The throughput of 1 server is 75.6 requests/sec in operation mix 1 and 80.1 requests/sec in operation mix 2, respectively.

1 distinct directory during every hour and the largest portion (17.9%) of requests were issued by clients who accessed 2 distinct directories during every hour. Requests are more scattered across bins in the histogram because larger bins have more accesses and hence more weights. We computed the histograms by dividing the traces into other time windows ranging from 30 minutes to 8 hours, but there was no significant difference across time windows.

Given the statistics of distinct directories, we compute the expected availability of the island-based file system for data, clients and requests, respectively, shown in Figure 4. Since the majority of web clients access a small number of distinct directories, the expected availability for this class of clients is high in spite of the fact that a partial failure in the system causes a random set of directories to be inaccessible. For example, if 1 out of 32 islands is down for an hour, we expect that 93.8% clients of the web server during that hour will not notice the temporary partial failure.

6.2 Single client performance

In this section, we present the results of running single client micro benchmarks on Archipelago in various configurations. The machines used in our experiments have Pentium II 300 MHz processors, 128 MB main memories and 6.4 GB Quantum Fireball IDE hard disks for use by Archipelago. The PCs are connected by a 3COM SuperStack II 100Mbps Ethernet hub. The PCs run Windows NT Workstation 4.0 and the hard disks for Archipelago are formatted in NTFS.

The set of micro benchmarks consists of 9 phases and each phase exercises one of the file system calls: CreateDir, SetDirAttr, CreateFile, SetFileAttr, ReadDir, WriteFile, ReadFile, DeleteFile and RemoveDir. The data set for the micro benchmarks is an inflated project directory that consists of 3600 directories, 3876 files and 154.4 MB of data in files. The 3876 files are stored in 540 directories and the rest of the directories are empty. Disk space is pre-allocated for each file in the CreateFile phase. The transferred block size in

the WriteFile and ReadFile phases is 64 KB or the file size, whichever is smaller. Each test is run more than 3 times and the results shown in this section are the averages.

We ran the micro benchmarks with a single client in five cases: directly on NTFS (NTFS), on the local machine of an Archipelago server (Local), on a remote machine from the server (Remote), with two servers (2 Servers), and with the consistency protocol turned on with two servers (Consistency), respectively. Figure 5 shows the bandwidth in WriteFile and ReadFile and the response times in other operations, all measured at the client side.

The difference between the NTFS and Local cases is caused by the overhead of computing hash functions. This overhead is low compared to the operation time itself. The difference between the Local and Remote cases is caused by the communication overhead (Win32 RPC on TCP/IP and 100 Mbps Ethernet) between the client and the server, i.e. 0.48 ms latency and 8.67 MB/s bandwidth in our experiments. There are two causes for the difference between the Remote and 2-Server cases: the cross-island operations such as CreateDir and SetDirAttr involve an additional server in the latter case and there was more total file system buffer cache in the latter case. The difference between the 2-Server and Consistency cases is caused by the overhead of the consistency protocol.

The results show that island-based design adds little overhead to NTFS and Win32 RPC performance and that the consistency protocol slows down the cross-island operations but does not have a noticeable impact on one-island operations.

We ran the same micro benchmarks with 1 to 16 servers and clients. The results, not shown here [14], indicate that the one-island operations scale linearly with the system and workload sizes. Two-island operations scale less efficiently and all-island operations do not scale because the consistency protocol requires $2*k$ uni-cast messages per cross-island operation, where k is the number of islands involved in the operation. Therefore, the overall scalability depends on the actual operation breakdown.

6.3 Operation breakdown in NTFS traces

Previous studies of file system traces indicated that the cross-island operations are rare [17] [9] [18]. However, it is well known that file access patterns are highly dependent on the operating systems where the traces were taken. Since we implement Archipelago on Windows NT as opposed to UNIX, in which the Sprite and NFS traces were taken, we feel it important to study the file access patterns in NTFS. We choose 7 workstations running Windows NT 4.0 and collected statistics on operations by running a trace program on each workstation. The users of the workstations include three graduate students, a software engineer, a home user and several lab users. The trace programs were run for 2 to 7 days and collected 30,391 to 480,385 total events. The trace program forks a thread to wait on each file system related event such as FileAdded through the NTFS event notification interface ReadDirectoryChangesW [19]. We present the events in Table 1 and infer the operation breakdown from them.

No.	Events	Avg.	Standard Deviation
1	Total Events	244408	140571
2	FileAdded	3.34%	1.70%
3	FileRemoved	2.38%	1.70%
4	FileRenamed	0.41%	0.31%
5	DirAdded	0.04%	0.07%
6	DirRemoved	0.03%	0.07%
7	DirRenamed	0.00%	0.00%
8	FileAttrModified	26.8%	10.8%
9	FileWritten	35.5%	11.3%
10	FileAccessed	16.3%	8.60%
11	FileSecurityModified	0.03%	0.04%
12	DirAttrModified	0.07%	0.07%
13	DirWritten	1.23%	1.59%
14	DirAccessed	13.9%	17.8%
15	DirSecurityModified	0.00%	0.00%
16	FileLinkModified	0.16%	0.08%
17	FileLinkRead	0.09%	0.10%
18	DirLinkModified	0.00%	0.00%
19	DirLinkRead	0.001%	0.002%

Table 1. Percentages of file system events in NTFS traces. Row 1 (Total events) shows the total number of events in all traces. Rows 2 through 19 show the percentage of each event. Shaded events correspond to cross-island operations in island-based file systems. The column "Average" shows the percentage of each event averaged over all traces. The column "Standard Deviation" shows the standard deviation of the percentages of each event

in each trace. Events not shown in the table have zero percentages. The names "FileLink" and "DirLink" refer to symbolic links (*shortcuts* in NT) to files and directories, respectively.

Table 1 shows that, on average, one-island operations account for 99.8% of total operations. The slow operations in island-based file systems, e.g. setting directory attributes, renaming directories, creating symbolic links (shortcuts) to directories, are rare.

The section below shows how the cross-island operations affect the overall scalability of the system, given the measured breakdown in this section.

6.4 Scalability of operation mixes

We run a benchmark of randomized operation mixes to measure the overall scalability of Archipelago. The benchmark is extended from the SPEC SFS or LADDIS benchmark [9]. Since Archipelago is implemented on top of NTFS, the operation mix in our benchmark uses NTFS API and is based on the operation breakdown we measured in NTFS, as shown in the previous section. The experiment environment and configuration are the same as in Section 6.2.

We ran the benchmark with 1 to 16 clients and servers on 1 to 16 machines. Each client runs on the same machine as a server, but accesses random files, directories and symbolic links across the entire system. The pre-created data set includes 2000 directories, 2000 files, and 100 symbolic links shared by all clients, and the same numbers of private objects (directories, files and symbolic links) per client. The client repeatedly does an operation that is randomly chosen at specified frequencies. For each operation, the client randomly chooses an object, either from the existing shared or private objects, or by generating a new name in an existing directory, depending on the operation. The WriteFile operation writes a random number (chosen from 0 to 1 MB) of bytes to the file; both WriteFile and ReadFile operations transfer up to 8KB per request so that the operation time is comparable to those of other operations. Each client maintains its own view of the shared objects and its private objects, but does not synchronize with other clients on the creation and deletion of the shared objects. Therefore, an operation on a shared object might fail if it conflicts with a previous operation on the same object from another client [9]. After the data set is pre-created, all clients run the randomized operation mix for 10 minutes. The throughput is calculated as the total number of successful operations by all clients divided by 10 minutes.

We ran the benchmark with two different operation mixes. Mix 1 exaggerates the cross-island operations and mix 2 is closer to the measured breakdown. The mixes cover a number of typical operations from each category, i.e. one-island, two-island and all-island. Note that more WriteFile than ReadFile events are recorded in the NTFS traces because reads that hit in cache cannot be captured by ReadDirectoryChangesW.

We recorded the actual client operations and server-to-server RPCs in the benchmarks, and estimated the speedups of the overall operation mix accordingly. Table 2 shows the recorded operation mixes and Figure 6 shows both the measured speedups and estimated speedups. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup with n servers is $n/(1+overhead\_per\_operation)$, where the *overhead per operation* is the total number of server-to-server RPCs divided by the total number of successful client operations.

	Mix 1 (%)	Mix 2 (%)
CreateDir	0.9297	0.0522
CreateFile	4.0314	3.5661
DeleteFile	2.7731	2.4353
DeleteLinkDir	0.9850	0.0128
ReadDir	14.4505	15.6528
ReadFile	14.1343	15.2778
RemoveDir	0.7543	0.0162
ResolveLinkDir	1.7205	0.1014
SetDirAttr	1.0383	0.0713
SetFileAttr	26.6085	29.2835
SymLinkDir	1.0089	0.0109
WriteFile	31.5656	33.5194
Successful	45360 to 309960	48042 to 756120
Total	48042 to 325534	48043 to 780260
Throughput (requests/sec)	75.6 to 516.6	80.07 to 1260.2

Table 2. Operation mixes. Each percentage in this table is the number of successful requests on each operation divided by the total number of successful requests, averaged over 1 to 16 clients and servers. The total numbers of requests and throughputs grow with the numbers of clients and servers for the fixed 10 minutes period; the ranges are shown in the last three rows in the table.

Operation mix 1 scales at a less than ideal slope due to the relatively large number of cross-island operations. For example, with 16 servers, the average overhead per operation is 0.8. The difference between the estimated speedup and measured speedup is due to the assumption of equal RPC processing times and local operation times. Load is well balanced across servers in both operation mixes; the largest/average requests per server are below 1.1 in all cases. Operation mix 2 is closer to the measured breakdown, i.e. contains a smaller number of cross-island operations; it scales nearly ideally in both estimated and measured throughputs.

6.5 Implications for larger scale systems

Given the percentages of one-island ($P1$), two-island ($P2$) and all-island (Pa) operations, where $P1+P2+Pa=1$, we can predict the speedup efficiency at large scale with an analytic model. Assuming that each local operation and RPC takes the same amount of time, the estimated speedup efficiency with n servers is $1/(1+overhead\_per\_operation)$, where the overhead per operation is the average number of server-to-server RPCs per operation and equals $(2-1)*2*P2+(n-1)*2*Pa$. (The factor 2 results from the two-phase commit protocol.) Two-island operations include CreateDir, RemoveDir, ReadFileLink and ReadDirLink; all-island operations include SetDirAttr, SetDirSecurity, SymLinkDir and RenameDir. Some operations, e.g. SetDirSecurity and SymLinkDir, did not show up in our statistical experiments; we inferred their percentages from other statistics [18]. The resulting percentages are $P1=99.768\%$, $P2=0.161\%$ and $Pa=0.071\%$. From the speedup efficiency model above, we predict that the system can scale up to 702 islands while maintaining the efficiency higher than 50%; that is, an island-based file system can achieve a higher speedup than 351 with 702 islands.

While such a large cluster is not currently available to us for experiments, our measurement results on the small cluster are encouraging and we are seeking external resources for further scalability tests.

6.6 Discussion

Although the target applications of an island-based file system are Internet services, we use a more generic benchmark in the scalability measurements. Our purpose of those measurements is to learn the impact of cross-island operations on the overall scalability of an island-based file system, but web access logs only give file-reading operations. We do not model in our benchmark the self similarity or hot spots in web accesses because it is not clear whether the same patterns will necessarily show up in disk accesses if web requests can be processed with data in the main memory cache of web servers or file system clients.

7. Related work

Existing file systems designed for high availability, such as Coda [28] and Ficus [29], replicate data across servers. Our approach in island-based file system, i.e. failure isolation, is complementary to the data redundancy approach for high availability. Client caching is extensively used in distributed file systems like Coda [28], Andrew [5] and Sprite [17] to support disconnected operations and to reduce traffics to servers. Similar to server replication, client caching improves availability by data redundancy, i.e. by replicating data in clients. It also improves scalability by reducing server load so that the same number of servers can serve a larger number of clients gracefully. Our scalability goal in island-based file system is to achieve efficient speedup when servers are added to the cluster, which is orthogonal to the goal of client caching. We have not implemented client caching in Archipelago, but we do not expect the island-based design to add any difficulty to such implementation.

State-of-the-art cluster file systems like Frangipani [1] and xFS [4] achieve high reliability and scalability by data redundancy. A fast system area network such as ATM is typically used in those cluster file systems for aggressive communications across data replicas. The majority of operations in island-based file systems do not require communication or synchronization across islands; therefore, an island-based file system can scale efficiently with commodity networks such as Ethernet. The ideal configuration for maximal reliability, availability and scalability is to run an island-based file system with a file system like Frangipani or xFS inside each island.

In terms of failure isolation, cross-node communications, locality and leveraging functions in local file systems, island-based file systems are comparable to distributed file systems like NFS [6], JetFS [12] and CIFS [11]. However, those systems do not share with island-based file systems scalability, load balance, and/or automatic data

partitioning and reconfiguration.

In Teradata [27], two orthogonal hash functions are used to map data items to two nodes. In an island-based file system, each data item is mapped to a single island but redundancy might be used inside the island. The Teradata approach offers better load balance when a single node fails, but the failures of two nodes always render a portion of data inaccessible. Our approach makes most operations involve a single island, isolates failures across islands, and does not lose data unless all replicas in the same island fail.

A large scale Internet service typically consists of three logical tiers: request distribution tier, service-specific processing tier and data storage tier. The Locality-Aware Request Distribution (LARD) [3] is a solution to locality and load balance in the distribution tier. The Cluster-Based Scalable Network Services (SNS) [21] [22] provides a programming model for the processing tier. In particular, the authors proposed application decomposition and orthogonal mechanism for graceful degradation during partial failures. Island-based design addresses failure isolation, locality and load balance in the storage tier. While the distribution-tier and processing-tier approaches suffice for read-mostly access patterns and weak consistency requirements, a robust and scalable storage tier is necessary for services with read-write access patterns and strong consistency requirements, such as shared calendar services and online shopping sites. The combination of the approaches in all three tiers can potentially achieve high availability and scalability for Internet services with a wide range of access patterns and consistency requirements.

Commercial web content distributors such as Akamai [24] [2] and Sandpiper [25] provide geographically distributed replication services to read-mostly web contents so that the latency in delivering contents to clients can be reduced. We are focused on improving the availability and scalability of local sites with read-write patterns. Their approach and ours are complementary to each other in improving the overall availability and scalability of Internet services.

Our main contributions are:

1. We address the availability and scalability issues for Internet services in the data storage tier.
2. Our approach to availability and scalability is isolating failures and reducing communication.
3. We achieve failure isolation and reduced communication by enforcing a one-island principle in hash-based data distribution and usage-based metadata replication.

8. Future work and conclusion

NT farms are a fact of life -- people are already using them to provide scalable services. An important question for people who are running all those NT farms to understand is how to structure the cluster in a way that can both balance loads and isolate failures without having to reinvent a distributed file system from scratch, which is a very difficult endeavor, by leveraging as much as possible from the existing NT infrastructure. Our experience suggests that this is indeed possible.

We designed an island-based file system as the data storage for highly available and scalable Internet services. We evaluated the design by statistical analysis of the access patterns in existing systems. We implemented Archipelago, a prototype of the island-based file system, and studied the performance of Archipelago in micro benchmarks and operation mixes.

We are considering extensions to the hashing of directories. Ideally, we would like to have an adaptive hashing algorithm that determines the height of a sub tree or the granularity of a file to hash based on the current state of load balance and access patterns. We are also going to improve the performance of all-island operations like SetDirAttr by replacing the $2*n$ unicast messages and $2*n$ replies with 2 broadcast or multicast messages and $2*n$ replies, where n is the number of islands.

We draw the following conclusions:

- The failure isolation provided by island-based file systems is useful to Internet services because a temporary partial failure can be made unnoticeable to the majority of clients.
- An island-based file system can scale well with the system and workload sizes because the majority of operations do not require communication or synchronization across islands.

9. Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, David C. Steere, for their valuable comments and suggestions. A number of theoreticians including Sanjeev Arora, Yaoyun Shi and Amit Chakrabarti helped us with the

mathematic modeling parts in the project.

References

[1] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System", in Proceedings of the 16th ACM Symposium on Operating Systems Principles, Octobor 1997.

[2] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", in Proceedings of the 29th ACM Symposium on Theory of Computing, May 1997.

[3] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers", in Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless Network File Systems", in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[5] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in A Distributed File System", in ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[6] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System", in Proceedings of USENIX Summer Technical Conference, Summer 1985.

[7] J. L. Carter, and M. N. Wegman, "Universal Classes of Hash Functions", in Journal of Computer and System Sciences 18, 1979.

[8] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files", in ACM Transactions on Database Systems, Vol. 4 No. 3, 1979.

[9] B. E. Keith, and M. Wittle, "LADDIS: the Next Generation in NFS File Server Benchmarking", in Proceedings of USENIX Summer Technical Conference, June 1993.

[10] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", in Proceedings of the 11th ACM Symposium on Operating System Principles, November 1987.

[11] Microsoft, the Common Internet File System (CIFS) specification reference, 1996.

[12] B. Gronvall, A. Westerlund, and S. Pink, "The Design of a Multicast-based Distributed File System", in Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999.

[13] H. Custer, "Inside the Windows NT File System", Microsoft Press, 1994.

[14] M. Ji, and E. W. Felten, "Design and Implementation of an Island-Based File System", Technical Report 610-99, Department of Computer Science, October 1999.

[15] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", in Communications of the ACM, July 1978.

[16] J. Gray, "Notes on Database Operating Systems", in Operating Systems: An Advanced Course, 1978.

[17] K. W. Shirriff, and J. K. Ousterhout, "A Trace-Driven Analysis of Name and Attribute Caching in A Distributed System", in Proceedings of USENIX Technical Conference, 1992.

[18] D. Roselli, and T. E. Anderson, "Characteristics of File System Workloads", Technical Report UCB//CSD-98-1029, 1998, and personal communications, April 1999.

[19] Microsoft Corporation, "Platform SDK: Windows Base Services: Files and I/O", in MSDN Library Visual Studio 6.0, 1998.

[20] Microsoft Corporation, "Windows NT IFS Kit", Early Release, March 1997.

[21] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services", in Proceedings of the 16th ACM Symposium on Operating Systems Principles, Octobor 1997.

[22] A. Fox, and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems", in Proceedings of HotOS-VII, March 1999.

[23] http://www.cs.princeton.edu

[24] http://www.akamai.com

[25] http://www.sandpiper.com

[26] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors", in Proceedings of the 15th ACM Symposium on Operating Systems and Principles, December 1995.

[27] DBC/1012 database computer system manual release 2.0. Technical Report Document No. C10-0001-02, Teradata Corporation, Nov 1985.

[28] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment", in IEEE Transactions on Computers 39(4), April 1990.

[29] G. J. Popek, R. G. Guy, T. W. Page Jr., J. S. Heidemann, "Replication in Ficus Distributed File Systems", in Workshop on the Management of Replicated Data, November 1990.

Single Instance Storage in Windows® 2000

by William J. Bolosky, Scott Corbin, David Goebel*, and John R. Douceur
*Microsoft Research, *Balder Technology Group, Inc.*
{bolosky, scottc, v-davidg, johndo}@microsoft.com

Abstract

Certain applications, such as Windows 2000's Remote Install service, can result in a set of files in which many different files have the same content. Using a traditional file system to store these files separately results in excessive use of disk and main memory file cache space. Using hard or symbolic links would eliminate the excess resource requirements, but changes the semantics of having separate files, in that updates to one "copy" of a file would be visible to users of another "copy." We describe the Single Instance Store (SIS), a component within Windows® 2000 that implements links with the semantics of copies for files stored on a Windows 2000 NTFS volume. SIS uses copy-on-close to implement the copy semantics of its links. SIS is structured as a file system filter driver that implements links and a user level service that detects duplicate files and reports them to the filter for conversion into links. Because SIS links are semantically identical to separate files, SIS creates them automatically when it detects files with duplicate contents. This paper describes the design and implementation of SIS in detail, briefly presents measurements of a remote install server showing a 58% disk space savings by using SIS, and discusses other possible uses of SIS.

1. Introduction

Some applications generate many files that have identical content. These files are separate from one another, in the sense that they may have different path names, owners, access control lists, and may charge different users' disk allocation quotas. Most importantly, because the files are separate, writes to one file do not affect any other files. However, the fact that the files have identical contents presents an opportunity for the file system to save space on the disk and in the main memory file cache. The Single Instance Store (SIS) is a pair of components in Microsoft® Windows® 2000 Server [Solomon 98] that automatically takes advantage of this opportunity. This paper describes the design and implementation of SIS, and discusses its use in Windows 2000 as well as other potential uses for the technology.

SIS is used in Windows 2000 to support the Remote Install Server [Microsoft 00]. Remote Install is an application that allows a server owner to configure a server with a set of machine images (installations of an operating system and an arbitrary set of applications), and to use these images to set up new machines quickly, without having to go through the usual installation processes. Different machine images stored on a remote install server will have different files, but there will be some duplicate files in the images. Running SIS on a remote install server removes space used by the duplicate files, and allows a server to host more images with the same amount of disk and main memory file cache space.

SIS is built in two components. First is a kernel-level file system filter driver (called the *SIS filter* or just the *filter*) that transparently implements files that have identical contents but are stored only once on the disk; second is a user-level service (called the *groveler*) that is responsible for automatically finding identical files and reporting them to the filter for merging. The filter's basic technique is to handle reads by redirecting them to common files, and to handle writes using copy-on-close. The groveler runs as a low-importance service (daemon) that tracks changes to the file system, maintains a database of hashes of files in the file system, checks files with matching hashes to see if they are identical, and reports matching files to the filter for merging. Unlike traditional file linking, SIS (via the groveler) automatically finds and merges files with duplicate contents. This is sensible in SIS because SIS links (unlike hard or symbolic links) are semantically identical to independent files.

SIS's copy-on-close technique is similar to copy-on-write, which has been used in various forms in computer systems for quite some time, most notably in virtual memory [Rashid 81] and database [Todd 96] systems. In copy-on-write, at the time of a "copy" a link between the source and destination is established, and the actual copying of the data is postponed until either the source or destination is modified. Crucial to the concept of copy-on-write is that it is semantically identical to a normal copy, unlike linked-file or shared memory techniques. SIS's copy-on-close differs from copy-on-write in that the copy is delayed beyond even the time of the first write until the complete set of updates are made to the

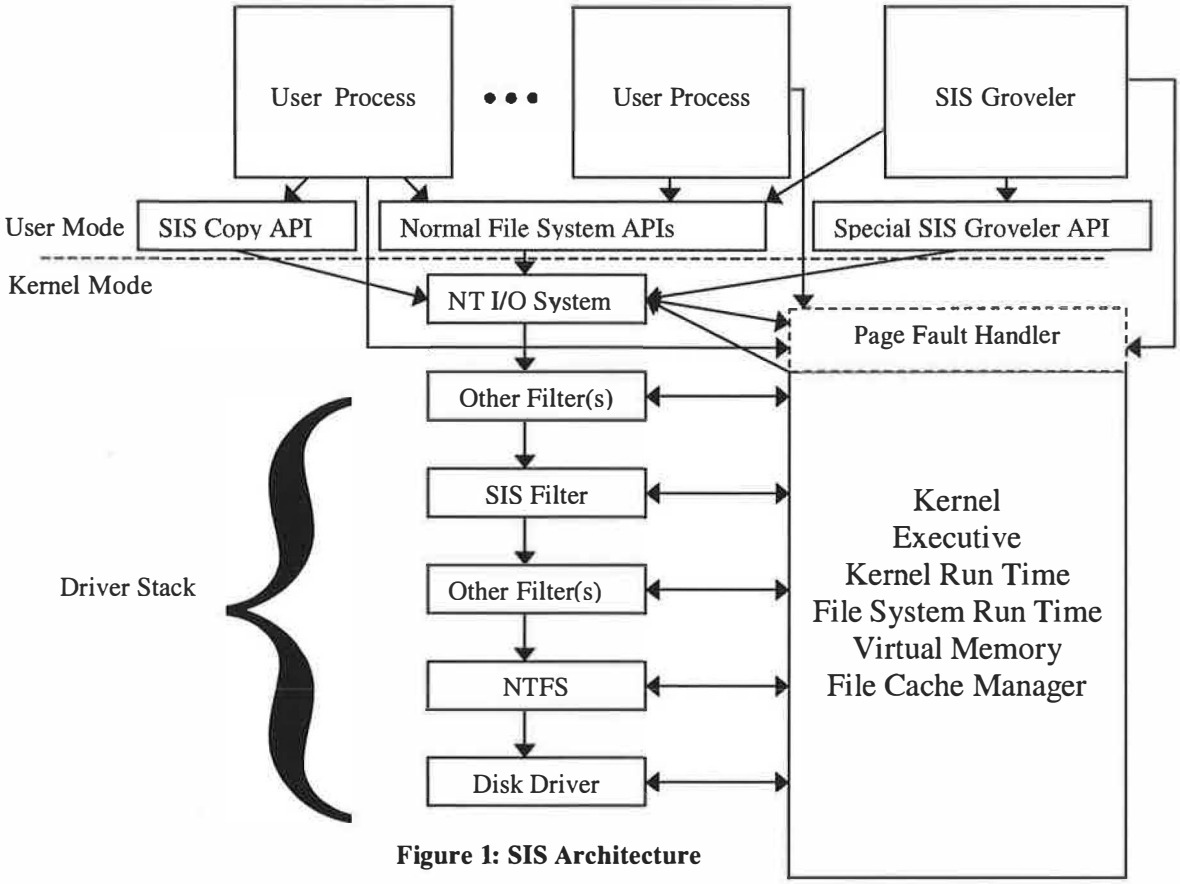


Figure 1: SIS Architecture

file, and then only the portions of the file that haven't been overwritten are copied. This has two advantages over copy-on-write: there is no delay for the copy at the time of the first write, and there is no need to copy the portions of the file that are overwritten.

While saving disk space is valuable, disk storage has been rapidly dropping in price per byte. Some may argue that the disk space savings afforded by SIS, which in most cases will be much less than an order of magnitude, are uninteresting for all but the most space-consuming applications. While this may be true, the reduction in size of the main memory file cache can have large performance effects. This will become more pronounced as the ratio of processor, memory and network speeds to disk latency increases.

The next section presents some background describing features in Windows 2000 on which SIS depends, and then goes on to describe SIS's architecture and implementation in detail. Section 3 briefly presents some performance measurements relating to the time and size overheads of SIS, and the disk space savings that was realized on a remote install server. The section then discusses some potential uses for SIS beyond the remote install server. Section 4 presents related work, section 5 summarizes and section 6 describes the availability of the software and raw data.

2. SIS Architecture and Implementation

SIS has two responsibilities: implementing SIS links, and discovering files with identical content in order to merge them. These two functions are provided by a kernel-level file system filter driver and a user-level service (daemon) respectively. Figure 1 shows the top-level architecture of SIS. This section lays out the basic concepts and terms used in describing SIS and its design, provides brief background on the Windows NT underpinnings used by SIS, and then describes the SIS architecture and design in detail.

A user file managed by SIS is called a *SIS link*. The SIS filter (the kernel mode portion of SIS) is responsible for assuring that users see appropriate behavior when accessing SIS links. The filter keeps the data that backs SIS links in files in a special directory called the *SIS Common Store*. SIS links may be created in two ways: A user may explicitly request a SIS copy of a file by issuing the SIS_COPYFILE file system control, or SIS may detect that two files have identical contents and merge them. The groveler (so called because it grovels through

the file system contents) detects duplicate files. All SIS functions are local to a particular NTFS volume, which may be accessed remotely in the same way as any other NTFS volume.

The rest of this section describes how SIS is designed and constructed. The first two subsections provide necessary background about unusual Windows 2000 and NTFS [Custer 94] facilities used in the implementation. Section 2.3 describes the SIS filter in detail, and section 2.4 covers the groveler.

2.1 The Windows 2000 Driver Model

The privileged mode portion of Windows 2000 consists of a base kernel containing services such as scheduling, virtual memory support, thread and multi-processor synchronization, etc.; and a set of loadable *drivers* [Solomon 98;Baker 97;Nagar 97]. Windows 2000 drivers may have more broad functionality than device drivers in traditional systems. They implement file systems, network protocol stacks, RAID/Mirroring disk functionality [Patterson 88], virus protection, instrumentation, off-line file migration, and other similar functions as well as simply operating normal devices [Fisher 98]. Groups of these drivers are called *stacks*, although that term can be somewhat misleading, because their organization is not necessarily linear (imagine a RAID driver that talks to more than one disk driver). A driver that is inserted between the NT I/O system and the base file system driver is called a *file system filter driver* (or sometimes just a *filter*). There are also other kinds of filter drivers that sit below the file system, but they are not relevant here.

To first approximation, NT driver stacks work by passing around I/O Request Packets (IRPs), which are requests to do a specific operation, such as read, write, open or close. For instance, a file system driver might receive a read IRP specifying that a range of a file should be read into a certain virtual address in a process. The file system would use its metadata to find the region(s) on disk holding the data for the given part of the file, modify the IRP to tell the disk driver what action to take, and then send the IRP down the stack to the disk driver, which would perform the actual I/O. When a driver marks an IRP as completed, any drivers above it on the stack have the opportunity to inspect the IRP, see if it completed successfully, and take action including aborting the IRP completion or changing the completion status.

The NT I/O manager sits at the top of all file system driver stacks. It accepts system calls from user processes (or function calls from kernel-level components), translates handles into pointers to file objects, generates IRPs and sends them to the top member of the appropriate driver stack. After the IRP completes, the I/O manager completes the initial request by completing the system or function call, or by other appropriate means in the case of asynchronous calls. The I/O Manager (and SIS) also support the FastIO function-call driver interface, but that is beyond the scope of this paper.

Windows 2000 supports memory mapped access to files. In this mode, a process or system component asks the system to map a portion of a file to a region of virtual memory. When the process or component accesses a virtual address in a mapped range, it may take a page fault, which will result in the virtual memory manager generating an IRP for reading the appropriate data into a page, and sending this request to the file system driver stack. This IRP is marked to indicate that it was generated by a page fault and that the file system should read data from the disk, rather than trying to obtain it from the system file cache. Once the file system completes the request, the virtual memory manager maps the page at the appropriate virtual address in the process or system address space and restarts the faulting thread. When an access to a memory mapped file doesn't result in a fault, the system is not immediately aware of it. This has an important implication for SIS: If a user maps a page, takes a read fault on the page and then later writes to it, SIS will not have any way of knowing that the page has been written, and so will not be able to take any consistency actions at the time of the write. Eventually, the virtual memory system will notice that the page is dirty and write it to disk, but the delay may be large. A different memory manager could mark such a clean, mapped page read-only and send a notification to SIS when a write happens (much like the technique used in distributed shared memory systems [Li 86] or Accent and Mach [Rashid 81; Accetta 86]), but there is no support for this in the NT memory manager.

The NT cache manager is a system component that maintains an in-memory cache of file contents. There is a single cache manager and single pool of cache pages for all of the different file systems and volumes on a particular Windows 2000 system. The cache manager operates by memory mapping files that are cached, and then using memory copy operations to/from the caller's memory in response to read/write calls. If the mapped page isn't present, the copy results in a page fault, which retrieves the appropriate data from the file system. The pages to which the cache manager has mappings are the same as the pages to which a user's memory mapped file would point, so memory mapped files and regular read/write based IO are coherent with one another. Misses in the cache result in page faults that are identical to those generated by user-level mapped accesses.

Because of the NT driver model, it is possible to develop filters with complex functionality independently of the other components with which they interact. The SIS filter was developed without any changes in NTFS, the NT I/O manager, virtual memory manager, cache manager or any other NT components.

2.2 Sparse Files and Reparse Points in NTFS

The Windows 2000 version of the Windows NT File System (NTFS) provides some new functionality that is used in the implementation of SIS: sparse files and reparse points. This section briefly describes these features.

A *sparse file* is a file that does not have physical disk space allocated for the entire file. Parts of the file that are not allocated are logically filled with zeroes. A file may be marked as sparse and extended without reserving disk space for the extension. An existing sparse file may have regions within the file deleted by a special IO control call, releasing the disk space and (logically) filling the deleted region with zeroes. A user can issue a different IO control that returns a description of the allocated and unallocated regions of a file. A write to an unallocated region causes disk space to be allocated. Unless a user specifically looks at a file to determine if it is sparse, it appears to be a normal file, possibly with much of the file being filled with zeroes. Users' disk quotas are charged for the sparse files as if they are fully allocated, regardless of how much disk space is actually used. Unallocated regions within files have a minimum granularity; the current implementation restricts them to aligned 64 Kbyte chunks.

A *reparse point* is a generalization of a symbolic link. A reparse point is placed on a file or directory by calling an IO control function. The reparse point consists of two parts: the reparse tag and the reparse data. The reparse tag is a 32 bit number that specifies the type of reparse point, and the reparse data is a variable size area that is not interpreted by the file system, but rather is used by a filter driver above NTFS that implements the functionality associated with the reparse point (or by the IO system in some special cases). SIS has a reserved 32 bit reparse tag.

When NTFS receives an open file request for a file with a reparse point, instead of doing a normal file open, it fails the request with `STATUS_REPARSE` and returns the reparse tag and data along with the completed (failed) IRP. Filters that use reparse points look for STATUS_REPARSE, and then check to see if the reparse tag is implemented by the filter. If not, the filter passes the completion up the driver stack. If the filter owns the tag, it can take whatever action is appropriate, based on the reparse data. If no driver claims the IRP and so the `STATUS_REPARSE` is passed all the way to the top of the driver stack, an error is returned to the caller.

There is an option flag bit for opens, `FILE_OPEN_REPARSE_POINT`, that specifies that reparse behavior should be suppressed. Unlike Unix symbolic links, a file with a reparse point on it is still an otherwise normal file. Specifying the flag tells NTFS that the file under the reparse point should be opened rather than returning a `STATUS_REPARSE` and letting the filters take action.

To illustrate the reparse point functionality, imagine implementing symbolic links using a filter driver and reparse points. The filter would have a reparse tag type allocated specifically for itself. The contents of the reparse data for a symbolic link would be the pathname component to be substituted for the file or directory in question. When the filter driver saw an open IRP complete with `STATUS_REPARSE` and the symbolic link tag, it would halt the completion process, modify the open request to have the pathname component from the reparse buffer replace the file name in the original open request, and send the request back to the file system for further processing. If an application wanted to create a symbolic link, it could simply place the appropriate reparse point on the file in question. To delete a symbolic link, it would open the link using `FILE_OPEN_REPARSE_POINT` (which would cause the open to complete without STATUS_REPARSE and thus prevent the filter driver from redirecting the open to the link target rather than the link) and then delete it in the normal way*.

2.3 The SIS Filter

The kernel portion of SIS is a filter driver that sits above NTFS. It handles all normal file operations that happen on SIS links, such as read, write, open, close and delete. It also implements a pair of special IO controls for creating new SIS links: `SIS_COPYFILE`, and `SIS_MERGE_FILES`. `COPYFILE` makes a SIS copy of a file, possibly turning the source file into a SIS link in the process. `MERGE_FILES` is used by the groveler to tell the filter to merge two files together. SIS_COPYFILE is unprivileged and is available to any user who has read permission to the source file and write permission to the destination. SIS_MERGE_FILES is privileged and only available to the groveler.

* Although the Win32 DeleteFile API [Microsoft 00a] takes a file name as its parameter, at the NT system interface level files are deleted by opening them and then sending down a delete call on the opened file handle.

The remainder of this section describes the details of SIS link files, implementing reads and writes, handling memory mapped accesses to SIS files, copy-on-close, volume check and backing up SIS links.

2.3.1 SIS Links

SIS links usually do not contain any file data, but rather contain a reference to another file called a *common store file*. Common store files contain the data for files managed by SIS, and are located in a protected directory. By having the data for SIS files located in the common store rather than in any particular link file, SIS avoids the problems that would arise when such a "primary" SIS file was deleted or overwritten.

A SIS link is implemented as a sparse file of the size of the file it represents with (usually) no regions allocated. Because there are no regions allocated, the file uses only as much space as is needed for its directory entry. A SIS link has a reparse point with a SIS tag. The contents of the data portion of a SIS reparse point are the name of a common store file that backs the contents of the link, a unique identifier for the link, a signature of the contents of the common store file backing the link, and some internal bookkeeping information. The purpose for the signature is described below.

Creation of a SIS link is fairly straightforward. A user issues a `COPYFILE` request. If the source file is not already a SIS link, its contents are copied to a newly allocated file in the common store, and the source file is converted into a link to that common store file. The destination file is then created as a link to the (either pre-existing or newly created) common store file. SIS keeps some out of band information (called *backpointers*) associated with the common store file that contains the set of links that point to the common store file. A COPYFILE request adds such a backpointer for the destination, and also for the source if it was not already a SIS link.

The reason that SIS copies the contents of a non-SIS file into the common store rather than renaming the file is that it is possible to open NTFS files by file ID, which is a number associated with the file, somewhat akin to a Unix i-number. When a file is renamed, its file ID stays the same. Therefore, if SIS renamed the source file into the common store, users of the file ID would attempt to open the common store file rather than the link file. By doing a real copy, the SIS filter avoids the problem, although this means that a SIS copy of a non-SIS file takes effort proportional to the size of the file. An extension to NTFS that allowed moving the contents of a

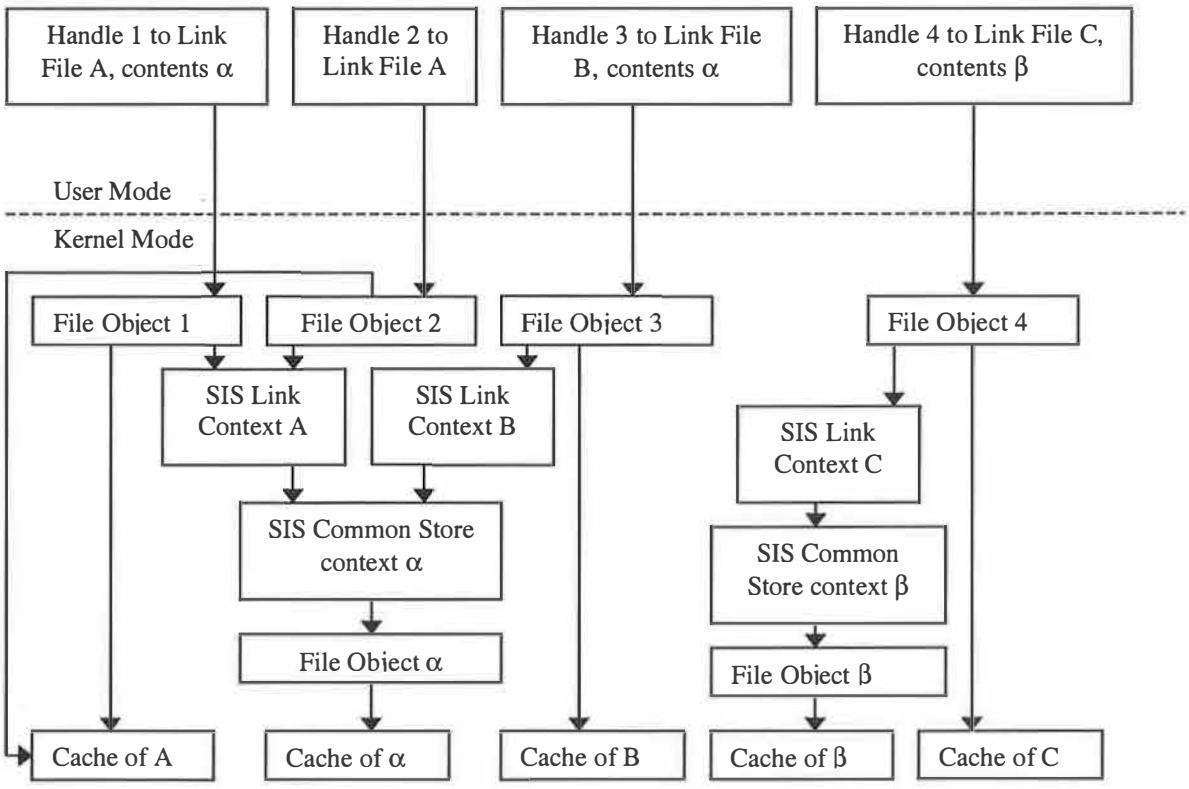


Figure 2: Example layout of SIS Contexts and Caching

file without moving its file ID would solve this problem, but it is not available at this time.

2.3.2 Normal Operations on SIS Links

When a user opens a SIS link, the filter intercepts the `STATUS_REPARSE` completion and resubmits the open request with the `FILE_OPEN_REPARSE_POINT` flag, resulting in the user's handle pointing to the link file (rather than the common store file). SIS also opens the common store file (if it has not already done so) and attaches some context to the user's handle indicating that this is a handle to a SIS link, and which common store file backs the link.

SIS must properly maintain coherency of SIS link files, meaning that users of a particular link must see all updates to that link, while seeing none of the updates to other links that are backed by the common store file. It is also desirable for SIS to conserve file cache space when possible by having different links to the same common store file backed by the same cache memory. Associating the cache with the common store file would violate coherence, while associating it with the link file would not conserve file cache space. Instead, SIS uses a hybrid of the two, where the caches associated with the links contain dirty (or potentially dirty) data and the caches associated with the common store files contain data that is known to be in common and clean.

Figure 2 shows an example of the in-memory structures created when a SIS link is opened. File objects are in-memory kernel objects that correspond to file handles, and are created by the kernel when handles are opened. Associated with a file object is a cache (maintained by the kernel cache manager), some file system state that is used to find the file's blocks on the disk (not shown), and (in the case of SIS links) a special context attached by SIS. The SIS link context contains a map of the file showing each region as clean, dirty or faulted. It also contains a pointer to a SIS common store context and some other various housekeeping information. The common store context contains a pointer to a file object that corresponds to the particular common store file, which in turn has its own cache. There is one handle and file object per successful CreateFile [Microsoft 00a] call, one SIS link context per open SIS link file (regardless of how many handles refer to it), and one common store context, file object and cache per common store file that has one or more open link(s) pointing to it.

As Figure 2 illustrates, there are two separate data caches associated with each opened SIS link: one for the link itself and one for the common store file that backs the link. In Figure 2, examples of these two caches for SIS link A are the boxes labeled "Cache of A" and "Cache of α" respectively. As long as there are no writes and no mapped file accesses, the cache associated with the link file is empty, and all of the cached file data is in the cache associated with the common store file. This is important in situations where more than one link to a particular common store file are is use at one time, because it reduces the cache requirements by up to a factor of the number of links to the common store file. This reduction in cache usage and the resultant ability to cache more file data in the saved memory is one of the main benefits of SIS to a file server.

Reads and writes initiated by ReadFile and WriteFile system calls (as opposed to by mapped file accesses, including those by misses by the system cache manager) are simple for SIS to handle. All writes are sent to the cache associated with the link file, and result in the written region of the file being marked dirty in the map in the SIS context associated with the link. When a ReadFile call tries to read a portion of a SIS file, the filter checks the map and sends the read to the common store file if the region is clean, and to the link file if the region is dirty or faulted. If the read spans multiple regions that are in different states, SIS splits the read into pieces and handles the pieces as appropriate.

2.3.3 Memory Mapped Access to SIS Links

Mapped file accesses present more of a challenge. When a user (or the system cache manager) maps a file and touches a page for the first time, it will generate a page fault which will be translated by the NT IO system into a read request that is sent to SIS. Read requests generated by page faults are specially marked, and the SIS filter can differentiate them from normal reads. Once SIS provides the data, the virtual memory system will map the appropriate page, and future accesses (either reads or writes) will not generate any action that will be seen by SIS. Other users who map and touch the same portion of the same file will be provided with a mapping to the same page, but the virtual memory system will not send a read to SIS, since it already has the data. The virtual memory system can unmap and throw away any clean (unwritten) pages without notifying SIS, and it will asynchronously generate writes for dirty pages at its own pace.

Because SIS cannot take any action when a mapped page is first written (because it doesn't get any notification of the write), in order to maintain coherence mapped pages must be associated with the link file cache and not the common store; if not, then two users could map two different SIS links that share a common store file, and they would see each others changes, violating the basic SIS semantics. We were unwilling to change the NT

memory manager to generate write faults when a page is first written.

Neither SIS's clean nor dirty states provides correct behavior in the case of a page that has taken a mapped read fault. Treating pages that have seen page faults as clean would result in normal (ReadFile) reads to those pages going to the common store file, which violates the coherence of normal reads with mapped writes. Treating them as dirty causes another problem: if there has been no write to the page the virtual memory system will see the page as clean, and may discard it. Sending a read to such a page would result in the user seeing a zero-filled portion of the link file, which is clearly wrong. To handle this problem, SIS has a third page state, *faulted*. When a page is in the faulted state, it sends normal reads to the cache associated with the link file, and page-fault reads to the common store file. Since a page-fault read will only happen if the page has been discarded by the virtual memory manager, when one occurs SIS concludes that the page has not been written and so it is safe to use the data from the common store file. SIS directs all accesses (reads and writes, page-fault and normal) to dirty file regions to the link file.

2.3.4 Copy-on-close

After all users close a SIS file that has had writes to it, the filter fills in the remaining clean regions (if any) with the data from the common store file. The actual copying happens in a system worker thread so as to not delay the close operation. If disk space is exhausted during this process, the filter simply leaves the remainder of the file unfilled, and leaves the reparse point intact. A subsequent open of this file results in the allocated regions of the file being marked dirty. We could have chosen to forego filling the clean regions of a file, and instead left the link backed in part by the common store file. If we had done so, we could have wound up in the odd situation of having SIS potentially increase the amount of disk space used in the system, by having files that are almost completely dirty backed by large and mostly unused common store files. To avoid this situation, SIS does the copy-on-close when possible.

Using copy-on-close rather than the more common copy-on-write has several advantages. The main one is that in many cases the file will be wholly overwritten, and so the copy can be avoided entirely. In his recent study of Windows NT 4.0 file system behavior, Vogels [Vogels 99] found that 79% of accesses to files were read only. Of the remaining accesses, 18% were write only and 3% were read/write. Of the write only file accesses, 78% of them were whole file writes. This means that of all files write accesses at least 67% were whole-file overwrites (conservatively assuming that none of the read/write files were wholly overwritten), and so copy-on-close would do no data copying. A second advantage is that the straightforward implementation of copy-on-write would stop all accesses to the file after the first write until the copy is completed, which could be a long time for large files. In an earlier paper [Douceur 99], we report that the distribution of files by size in Windows file systems is heavy-tailed, and so there are significant numbers of very large files; both we and Vogels found that about 20% of bytes are in files 4MB and larger, so it is reasonable to believe that writes to large files would not be unusual, and copy-on-write delays unacceptable. A final advantage of copy-on-close is that it allows sharing of file cache space for file regions that are unwritten.

2.3.5 Implementation Details and Backup

When a SIS link is eliminated, either by deletion of the link or because of an overwrite, the filter removes the corresponding backpointer in the common store file. When all of the backpointers for a common store file are removed, the filter deletes the common store file.

In certain circumstances, it is impossible for the SIS filter to prevent a user from writing an arbitrary reparse point. In theory, the user could write a SIS reparse point, which the filter would then use to read data for the user from the common store. This would be a violation of security, since the user did not have to prove access permission for the common store file before writing the reparse point. In order to address this problem, SIS includes a signature of the common store file contents in the reparse data. The signature is a hash of the entire contents of the common store file; it is easy to compute given the data in the file, but impossible to compute without the contents of the file (and 64 bits in length so randomly guessing is difficult). Each link file contains a copy of this signature in its reparse point. The filter will refuse to open a link file that contains an invalid signature. A reparse point that contains an appropriate signature proves that the user already knows the contents of the common store file (or has seen another reparse point that refers to the same contents, and so could have known the contents of the file). Because all that a user gets by creating SIS link is the ability to read the common store file contents, there is no security breach.

The SIS filter includes a facility called *volume check* for repairing inconsistencies in metadata, similar to the Unix `fsck` and Windows `chkdsk` programs. Unlike these programs, however, SIS is able to do its repairs while the system is running, without having to disrupt service in any major way. One limitation during a volume check is that if a user deletes what appears to be the last reference to a common store file the filter will not delete the common store file until the volume check completes,

because it does not trust its backpointers during a volume check. The volume check process will repair the backpointers for all common store files, and will delete any common store files that have no references. NTFS provides a method for efficiently finding all SIS links on a volume, so the time to complete a volume check is proportional to the number of SIS links on the volume, not to the size or total number of files on the volume. A volume check is initiated any time that the SIS filter notices an inconsistency.

SIS provides a special interface to backup/restore applications that allows them to behave appropriately with SIS links [Bolosky 99]. The goal is to have exactly one copy of the SIS file content on the backup tape for each backed up set of SIS links that refer to a given common store file. SIS provides a dynamically loaded library (DLL) for the backup/restore application. The backup application calls the DLL when it encounters a SIS link, and the DLL tells backup if it needs to back up a common store file in response. On restore of a SIS link, restore calls the DLL, which in turn looks to see if the appropriate common store file already exists or if it's already reported that file to restore. If not, then it reports the common store file corresponding to the link being restored. Because common store files have universally unique file names, and their content never changes once the file is created, if the common store file still exists on the volume there is no need to restore over it; simply linking to it suffices.

2.4 The Groveler

The groveler is a user level process that finds duplicate files in the file system, and reports these files to the SIS filter for merging. The essence of its task is efficiently to find the sets of matching files on a volume, and to keep the sets up to date as the volume changes. It maintains a database of signatures of files on the volume, and uses NTFS 5.0's update journal feature* to track files that have changed and to update their database entries.

The groveler database contains two structures: a work queue, and a mapping of signatures to files. The work queue contains work items of two types. The first is to compute the signature of a file, and the second is to compare a file to any others with matching signatures. The groveler has a thread that runs periodically, reads entries from NTFS's update journal and if appropriate makes entries on the groveler's work queue. If the groveler misses entries in the update journal, it detects this fact and re-scans the entire file system. A second thread drains items from the work queue, either computing a signature or comparing a pair of files for each item. It then updates the database, possibly instructs the filter to merge two files, and removes the item from the queue.

The groveler uses a 128 bit file signature. The first 64 bits of the signature are the size of the file. It is inexpensive to obtain the file size, and files with differing size obviously cannot be identical. The remaining 64 bits are computed by running a hash function on a fixed portion of the file's contents. The groveler hashes two 4 kilobyte chunks of file contents from the middle of the file (unless the file is less than or equal to 8 kilobytes in size, in which case it hashes the entire file). Hashing only part of the file means that any differences in the portion of the file not hashed will not be detected by comparing the signatures. However, restricting the hash limits the amount of work that the groveler does for large files. Even if the groveler hashed all of the file, the possibility of hash collisions would still exist, and in order to guarantee that it does not corrupt data it would be required to compare files byte-by-byte before linking them together. Therefore, unless hashing a smaller portion of a file greatly increases the probability of hash collisions for unequal files, the optimization is worthwhile. We have noted very few cases wherein files' hashes match and the files' contents do not, even though we only hash a constant amount of data per file.

The groveler is set up to run as a low importance background task (unless disk space is tight). The groveler is not particularly CPU intensive; most of the work it does is disk I/O. As a result, simply running it at low scheduling priority does not prevent it from interfering with other, more important tasks. The groveler uses a scheme where it tracks its own rate of progress, and slows down its run rate when its progress slows. The premise is that if the groveler is running more slowly, it must be contending with some other, higher priority process for some resource. The toolkit that implements this technique is called "MS Manners" and is described in detail in [Douceur 99a].

In order to do its work the groveler needs to open files, and in some cases to hold them open for a long time. This could cause problems for other users of the files who might want to have exclusive access to a file for some reason, and would have their opens fail with a sharing violation. In order to mitigate this problem, the groveler takes a batch oplock on each file it opens immediately after opening it. Oplocks [Microsoft 00a]

* The update journal [Microsoft 00a] is a feature whereby NTFS maintains a record of all recent updates to a volume in a ring buffer. Each entry in the journal has a sequence number (USN, "update sequence number"), so it is possible for a user of the journal to determine if it has missed any entries because of a gap in the sequence numbers.

are a facility by which the opener of a file can be notified when another user wants access to the file, and can take action before the other user's action is completed or failed. They were designed to allow effective local caching of files that are shared over a network while maintaining coherence. However, in the case of SIS they allow the groveler to close a file that it is using without generating a sharing violation for another user of the file.

3. Performance of and Uses for SIS

In Windows 2000, SIS is deployed only on remote install servers, and then only on the volumes that contain the remote install system images. While the initial release of SIS is used only for remote install servers, we believe that in the future it may be put to other uses. We briefly present measurements that show that the per-link space overhead of a SIS link and the time to make a copy of a SIS link. We measured a remote install server at Microsoft, and report on the disk space (though not file cache) savings that SIS provided for that server. We also discuss using SIS on file servers that back the files of multiple users, and using SIS in a serverless, distributed file system that we have proposed elsewhere [Bolosky 00].

We measured the time to make a SIS copy of a file that is already a SIS link using the SIS_COPYFILE API on a Gateway 2000 E-5000 Pentium II 300MHz personal computer running Windows 2000, with 512 MB of memory, and a 9 GB ST39173N 7200 RPM Seagate Barracuda disk drive containing an aged file system and a 9 GB ST19171W Seagate Barracuda 7200 RPM disk drive containing a clean file system. We ran all tests with the network disconnected. The clean file system was almost empty and newly formatted while the aged file system was 70% full and had been in use for several years, although it had been defragmented a few months before we ran the test.

We determined the time to make a SIS link by creating a file, making one SIS copy of it to make the source file a SIS link, making 10,000 SIS copies of the file, noting the wall clock time for the execution and dividing by 10,000. We ran the 10,000 copy measurement 100 times on each of the two file systems. In each instance, we discarded the first 10,000 copy run so that we would be running with the file caches hot.

In our tests (using a file about 1.6MB in size), a SIS copy took 4.3ms (+/- 290µs at 99% confidence) on the clean file system and 8.6ms (+/- 220µs at 99% confidence) on the used file system to copy the file. It is difficult to say how much of the difference is due to the state of the file system and how much is due to the fact that the dirty file system is running on an older model of disk. The reason that the copy is this slow is that there is a synchronous disk write in the SIS backpointer update, which is necessitated by the inability of several NTFS metadata updates to be grouped into a single atomic transaction. Copying the same 1.6MB file using a normal file copy took about 260ms per copy on the clean file system.

We also measured the space cost to create a SIS link by noting the amount of free disk space, making a large number (10,000) of SIS links, noting the amount of free space again, and dividing by the number of links. The overhead was about 300 bytes/link for all power-of-ten file sizes from 10^0 to 10^8 bytes.

We measured a remote install server at Microsoft that is used to install various versions of Windows NT for testing purposes (of the systems being installed, not the server). This server contains 20 different images of Windows NT of various flavors: both Windows 2000 Professional and Server, and different internal builds of the system, including the last five builds before the final Windows 2000 product, the release version of Windows 2000, and a build from after the Windows 2000 release. The remote install volume on this server contained about 112,000 files, and a logical 7.5 GB of file contents. Of this, 45,000 files (39%) and 6.0GB (80%) were in SIS links, backed by 1.6GB in 13,000 common store files. The overall space savings from SIS was 58%. In a field deployment, one would expect to see fewer different versions of the operating system with more different sets of applications installed, which we expect would result in better space savings.

We would have liked to measure the advantage in buffer cache usage for a remote install server running SIS. Unfortunately, we did not have the ability to take traces of a real remote install server in action. While we could have used synthetic workload traces to simulate such a server, barring real measurements there is no good way to determine the parameters for the workload generator. Because cache performance is strongly (and non-linearly) influenced by the working set size, the exact workload parameters would almost wholly determine the results of such a simulation. Therefore, lack of real parameters makes the synthetic workload exercise at best meaningless, and more likely misleading.

SIS could be used for file servers that store the files of groups of users, such as is typically done with NFS [Pawlowski 94; Sandberg 85], AFS [Howard 88], Coda [Satya 90; Kistler 91] or any of a number of comparable systems. In fact, the original reason for building SIS was to support the remote boot server, a similar idea for Windows 2000, but remote boot was cut from the final product for schedule reasons.

In [Bolosky 00] we measured the contents of a number of desktop personal computer file systems at Microsoft and observed the duplication of contents among them. We found that the level of duplication depends on the number of file systems grouped together, with the percentage of reclaimable space growing roughly with the log of the number of file systems. Grouping 100 randomly selected file systems gave a little better than 30% space savings. At 1000 file systems the savings was just under 50%. Our model predicts a little better than 60% savings at 10,000 file systems. File systems of people with similar job functions (e.g. software developer, secretary, manager) were more alike than the randomly selected groups of file systems quoted above. We are unaware of any comparable study or raw data for any other environment, but we expect that there may be significant differences on other operating systems or even for institutions other than Microsoft.

The file system that we propose in [Bolosky 00] is intended to provide a shared name space and common access to storage for tens of thousands of users using only the desktop workstations of those users. One issue in such a design is providing availability of files on machines that are much less available than managed servers. Our approach is to replicate the file contents across the machines in the hope that the system can find at least one copy of a file on a machine that is up. The number of replicas that can be made strongly (exponentially) influences the probability that a file will be available. By using SIS (or a similar technique) the total size of the stored file content can be reduced, and hence the number of copies that will fit on the available disk space will be increased, greatly improving the overall availability of the system. That is, even if SIS only provides modest space savings, these savings can result in greatly improved system performance.

4. Related Work

There are a number of different uses of copy-on-write in computer systems. Mostly they share the same characteristics: A traditional copy would be expensive in time or space (or both); the semantics presented to the user are those of a copy, rather than a link; and, the expectation of the system designer is that the copy-on-write will rarely happen. Typically, the cost of the initial "copy" followed by the copy-on-write is higher than just eagerly evaluating the copy, but this is made up for by the common case in which the copy-on-write never happens [Fitzgerald 86].

Copy-on-write has been used in virtual memory systems as least as far back as Accent [Rashid 81] and Mach [Accetta 86;Young 87]. These systems allowed processes (including file systems) to send messages to one another with copy semantics, but used the virtual memory system to map the same memory into both processes' address spaces. If a process subsequently wrote into the memory, the system took a protection fault, made a copy of the page in question, and mapped the newly copied page into the faulting process's address space with write permission.

The Microsoft Exchange Server [Todd 96] (a multi-user mail server) uses copy-on-write techniques for mail messages that are sent to multiple recipients, and even calls the technique "Single Instance Store." This mail system allows recipients to modify mail messages after they're received, which triggers the copy-on-write. It does not have an equivalent of the groveler; if two identical messages are in a server, but they were not generated as copies of one another, the server will never merge them.

Apollo systems used access control lists (ACLs) that described the lists of users who had access to particular files. These ACLs were immutable. The `salacl` command looked through the sets of ACLs existing in the system, and combined those that matched [Leach 98], functionality that is similar to the groveler, but the because the ACLs were immutable there was no need for the copy-on-close function of the SIS filter.

Many file systems support various types of links with semantics differing from SIS links. In particular, Unix file systems [McKusick 84] typically support both hard and symbolic links. These types of links differ from SIS links in that writes through one link to an object are visible through a different link. NTFS [Custer 94] supports hard links that are similar to those in Unix. None of these systems automatically detect and merge files with identical contents, because such an action does not make sense when links have different semantics from separate files.

It is common practice to aggregate files of many users onto a central file server, which may be implemented as a single machine or a cluster. NFS [Sandberg 85], Sesame [Thompson 85], AFS [Howard 88], the Sprite file system [Ousterhout 88], Coda [Sayta 90], Ficus [Guy 90] , Swift [Cabrera 91], Zebra [Hartman 93], the Microsoft file systems using the Server/Redirector network remoting services [Solomon 98], and a host of others too numerous to mention all are variations on the theme of a centralized network file server that often will combine the files from many users on many client machines onto a single file system.

There are number of serverless (ie., decentralized) distributed file systems similar to the one that we mention at the end of Section 3. Chief among them are

Frangipani [Thekkath 97], and xFS [Anderson 95]. They differ from our proposed file system in a number of ways, among them that they assume that the machines that implement the system are secure, and are more highly available than is reasonable to expect from desktop workstations.

5. Summary

SIS is a component of Windows 2000 that detects files that have identical contents and merges them into SIS links, special links that present the semantics of separate files while in most cases using the disk and file cache space of a single file (plus a small disk overhead per link that does not depend on the size of the underlying file). In Windows 2000 SIS is used as part of the remote install server, which is a way of setting up machines to a pre-determined configuration without having to go through the normal set-up process.

SIS is implemented as a file system filter driver and a user-level service. The filter driver implements the links and copy-on-close when a file is modified, presenting the semantics of an independent copy to the user of the link. The user-level service watches changes to the file system, computes signatures for newly-changed files, compares files with matching signatures and reports matching files to the filter for merging.

The cost of making SIS copies of files that are already SIS links is small and independent of the size of the file. The disk-space overhead of a SIS link is about 300 bytes regardless of the size of the file to which the link refers. In most instances, copying a SIS file takes about 8.6ms on the (pretty slow) machine and thoroughly aged file system on which we took our measurements. We were unable to measure the impact of SIS on file cache usage in a real installation, but hypothesize that in some cases it may significantly improve performance by reducing the cache working set below the available memory size.

We speculate that SIS could be useful in contexts other than the remote install server, in particular a distributed, serverless file system built on ordinary workstations. Because that file system's performance is strongly influenced by the amount of free disk space available, using SIS to reduce the effective disk space usage provides large benefits in availability.

6. Availability

SIS ships with Microsoft Windows 2000 as part of the Remote Installation Services. The source code is available with a Windows 2000 source license, which is available from Microsoft on a case-by-case basis. A suitably sanitized version of the raw data used in [Douceur 99] and [Bolosky 00] is available by request from John Douceur, johndo@microsoft.com, on a set of CD ROMs. It is much too big to place on the net for download.

Acknowledgments

We would like to thank Cedric Krumbein, Mihai Popescu-Stăneşti, Drew McDaniel, Steven West, Galen Hunt and Yi-Min Wang for their help in implementing and testing the SIS components. Some of the early conceptual work on SIS was done in conjunction with Rick Rashid, Nathan Myhrvold and Terri Watson Rashid. Rob Short and Chuck Lenzmeier were invaluable in navigating the politics of the Windows NT group. Brian Andrew provided help in understanding NTFS and its behaviors, and provided useful suggestions as to how to implement SIS. Steven West and Matthew Stevens provided access to remote install servers so that they could be measured. Felipe Cabrera commented on the SIS/backup interactions.

References

[Accetta 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for UNIX Development," In *Proceedings of the Summer USENIX*, July, 1986.

[Anderson 95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. "Serverless Network File Systems," In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 109—126, December 1995.

[Baker 97] A. Baker. *The Windows NT Device Driver Book*, Prentice Hall PTR, 1997.

[Bolosky 99] W. Bolosky. "The SIS/Backup Interface," available upon request from Steve Olsson, solsson@microsoft.com.

[Bolosky 00] W. Bolosky, J. Douceur, D. Ely and M. Theimer. "Evaluation of Desktop PCs as Candidates for a Serverless, Distributed File System," to appear in *Proceedings of ACM SIGMETRICS 2000*.

[Cabrera 91] L. Cabrera and D. D. E. Long. "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, 4(4):405—436, Fall 1991.

[Custer 94] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[Douceur 99] J. Douceur and W. Bolosky. "A Large-Scale Study of File System Contents," in *Proceedings of ACM SIGMETRICS '99*, pp. 59—70, May 1999.

[Douceur 99a] J. Douceur and W. Bolosky, "Progress-based regulation of low-importance processes," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 247—260, December, 1999.

[Fisher 98] L. Fisher. *The Windows NT Installable File System Kit.* A Microsoft product that can be ordered from http://www.microsoft.com/hwdev/ntifskit.

[Fitzgerald 86] R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, 4(2):147—177, May, 1986.

[Guy 90] R. G. Guy, J. S. Heidemann, W. Mak, T. W. P., Jr, G. J. Popek, and D. Rothmeier,. "Implementation of the Ficus Replicated File System," Proc. of the Summer 1990 USENIX Conference, pp. 63—71, June, 1990.

[Hartman 93] J. H. Hartman and J. K. Ousterhout. "The Zebra Striped Network File System," *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 29—43, December, 1993.

[Howard 88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1):51—81, February 1988.

[Kistler 91] J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System," In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 213—225, October, 1991.

[Leach 98] P. Leach. Personal communication. He said this is only documented in the manuals, not any other published sources, and these manuals are now hard to find.

[Li 86] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems," In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pp. 229—239, August 1986.

[McKusick 84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. "A Fast File System for Unix," *ACM Tranactions on Computer Systems*, 2(3):181—197, August, 1984.

[Microsoft 00] Microsoft Windows 2000 Server online help file. Microsoft Corporation, February 2000.

[Microsoft 00a] Microsoft Developer Network Library. A product available from Microsoft. See http://msdn.microsoft.com/. January, 2000.

[Nagar 97] R. Nagar. *Windows NT File System Internals.* O'Reilly, 1997.

[Ousterhout 88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch. "The Sprite Network Operating System," *IEEE Computer* 21(2):23—36, February, 1988.

[Patterson 88] D. A. Patterson, G. Gibson, and R. H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)," In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pp. 109—116, June 1988.

[Pawlowski 94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel and D. Hitz. "NFS Version 3 Design and Implementation," In *Proceedings of the Summer USENIX Conference*, pp 137—152, June 1994.

[Sandberg 85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and Implementation of the Sun Network Filesystem," In *Proceedings of the Summer USENIX Conference*, pp. 119—130, June 1985.

[Rashid 81] R. Rashid and G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel," In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 64—75, December, 1981.

[Satya 90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. "Coda: A Highly Available Filesystem for a Distributed Workstation Environment," *IEEE Transactions on Computers*, 39(4), April 1990.

[Solomon 98] D. Solomon. *Inside Windows NT, Second Edition.* Microsoft Press, 1998.

[Thekkath 97] C. Thekkath, T. Mann and E. Lee. "Frangipani: A Scalable Distributed File System," In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 224—237, December, 1997.

[Thompson 85] M. R. Thompson, R. D. Sansom, M. B. Jones, and R. F. Rashid. "Sesame: The Spice File System," Carnegie-Mellon University Computer Science Technical Report CMU-CS-85-172, Carnegie-Mellon University, Pittsburgh, PA. 1985

[Todd 96] G. Todd, *et al. Microsoft Exchange Server Survival Guide.* Sams Publishing, 1996.

[Vogels 99] W. Vogels. "File system usage in Windows NT 4.0," In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 93—109, December, 1999.

[Young 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 63—76, November, 1987.

User-level Resource-constrained Sandboxing

Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{fangzhe, ayali, vijayk}@cs.nyu.edu, http://cs.nyu.edu/pdsg

Abstract

The popularity of mobile and networked applications has resulted in an increased demand for execution "sandboxes"—environments that impose irrevocable restrictions on resource usage. Existing approaches rely on kernel modification for enforcing quantitative restrictions (e.g., limiting CPU utilization of an application to 25%). However, the general applicability of such approaches is constrained by the difficulty of modifying shrink-wrapped operating systems such as Windows NT.

This paper presents a user-level sandboxing approach for enforcing quantitative restrictions on resource usage of applications. Our approach actively monitors an application's interactions with the underlying system, proactively controlling them to enforce the desired behavior. Our approach leverages a core set of user-level mechanisms that are available in most modern operating systems: fine-grained timers, monitoring infrastructure, debugger processes, priority-based scheduling, and page-based memory protection. We describe implementation of a sandbox on Windows NT that imposes quantitative restrictions on CPU, memory, and network usage. Our results show that application usage of system resources can be restricted to within 3% of desired limits with minimal run-time overhead.

1 Introduction

The increasing availability of network-based services and the growing popularity of mobile computing has resulted in an increased demand for execution "sandboxes"—environments that support differentiated service and impose irrevocable restrictions on resource usage. For instance, the execution environment can ensure *qualitative* restrictions such as permitting an application component to only access certain portions of the file system (e.g., `c:\temp`), and *quantitative* restrictions such as limiting the component to 20% of CPU share. These qualitative and quantitative restrictions isolate the behavior of other activities on the system from a potentially malicious application component, and are desirable for the wider deployment of distributed component-based applications.

Existing approaches for enforcing qualitative and quantitative restrictions on resource usage rely on kernel support [JLDB95, MST94], binary modification [WLAG93], or active interception of the application's interactions with the operating system (OS) [BG99, ET99, GWTB96]. The kernel approaches are general-purpose but require extensive modifications to OS structure, limiting their applicability for expressing flexible resource control policies. The remainder of the approaches rely on deciding for each application interaction with the underlying system whether or not to permit this interaction to proceed; consequently, they provide qualitative restrictions (such as whether or not a file-reading operation should be allowed), but are unable to handle most kinds of quantitative restrictions, particularly since usage of some resources (e.g., the CPU) does not require explicit application requests.

This paper presents a user-level sandboxing approach for enforcing quantitative restrictions on application's resource usage. Our approach actively *monitors* the application's interactions with the underlying system, proactively *controlling* them to enforce the desired behavior. Our strategy recognizes that application access to system resources can be modeled as a sequence of requests spread out over time. These requests can be either implicit such as for a physical memory page, or explicit such as for disk access.[1] This observation provides two alternatives for constraining resource utilization over a time window: either control the resources available to the application at the point of the request or control the time interval between resource requests. In both cases and for all kinds of resources, the specific control is influenced by the extent to which the application has exceeded or fallen behind a *progress metric*. The latter represents an estimate of the resource consumption of the application program.

[1] Those disk operations incurred by paging are also implicit.

For this approach, the primary challenge lies in accurately estimating the progress metric and effecting necessary control on resource requests with acceptable overhead. It might appear that appropriate monitoring and control would require extensive kernel involvement, restricting their applicability. Fortunately, most modern OSes provide a core set of user-level mechanisms that can be used to construct the required support. Presence of *fine-grained timers* and *monitoring infrastructures* such as the Windows NT Performance Counters and the UNIX `/proc` filesystem provides needed information for building accurate progress models. Similarly, fine-grained control can be effected using standard OS mechanisms such as *debugger processes*, *priority-based scheduling*, and *page-based memory protection*.

We describe the implementation of a sandbox using the above strategy to impose quantitative restrictions on three representative resources—CPU, memory, and network on Windows NT (The same approach has also been used to implement sandboxing on Linux). A detailed evaluation shows that our implementation is able to restrict resource usage of unmodified applications to within 3% of the prescribed limits with minimal runtime overhead. We also present a synthetic application that demonstrates the flexibility of a user-level sandbox. In this case, our approach permits application-specific control at fine granularity---over differentiated thread and socket groups.

The rest of this paper is organized as follows. Section 2 describes background and related work. Section 3 presents the overall sandboxing strategy and discusses its application for three example resource types: CPU, memory, and network. The concrete implementation of the sandbox on Windows NT is presented and evaluated in Section 4. Section 5 highlights the flexibility of user-level sandboxing, and we conclude in Section 6.

2 Background and Related Work

The problem of ensuring that application components are guaranteed a required level of service and do not violate certain qualitative and quantitative restrictions on resource usage has recently attracted a lot of attention. Related approaches can be classified into two broad categories: kernel-level mechanisms and code transformation techniques.

Kernel-level mechanisms Real-time Mach supports a *Capacity Reserve* abstraction [MST94] that guarantees applications a predictable CPU share over periodic time interval. Rialto [JLDB95, JR99] introduces *CPU Reservation* and *Time Constraints*, extending the NT kernel to support real-time applications. Resource containers [BDM99] proposes a new UNIX kernel model for accounting and scheduling resources, which enables fine-grained and predictable resource allocation. Eclipse [BGOS98] implements reservation-domain scheduling of multiple resources (CPU, disk, and physical memory). Resource Kernels [RJMO99] guarantees an application's timeliness requirements and disk bandwidth using classified reservation schemes. All these approaches require extensive modifications to OS structure. Consequently, their applicability for implementing flexible resource control policies is limited, particularly for shrink-wrapped OSes such as Windows NT.

In addition, primarily in the context of real-time operating systems, several scheduling algorithms have been proposed for constraining/fair-sharing CPU resources (e.g., Stride scheduling [WW95], Lottery scheduling [WW94], SMART scheduler [NL97], and Start Time Fair Queuing [GGV96]) and network resources (e.g., Weighted Fair Queuing [DKS89] and Virtual Clock [Zha91]). Implementing these algorithms in a user-level scheduler on Windows NT is restricted because of the interference from OS-level scheduler.

Restricted versions of such mechanisms are also available in the form of job control mechanisms [Ric99] in Windows 2000. The latter allows expression of constraints on resource limits for process groups (e.g., maximum total execution time). We complement the job control scheme using a flexible user-level approach, which additionally provides support for constraining network bandwidth and weighted fair-sharing of CPU resources.

Code transformation techniques provide a user-level approach for imposing restrictions on resource usage. These techniques, which include binary modification approaches (such as software fault-isolation [WLAG93]) and API interception approaches (such as Janus [GWTB96], Mediating Connectors [BG99], and Naccio [ET99]), all rely on monitoring an application's interactions with the underlying OS. These techniques leverage OS mechanisms such as system-call interception by a debugger process [GWTB96], or application structuring mechanisms such as DLL import-address-table rewriting [BG99, HB99] to execute some checking code whenever the application interacts with the OS. This code decides, for relevant interactions, whether to allow, delay, or deny the interaction from proceeding.[2]

Consequently, such approaches provide the necessary hooks for enforcing qualitative restrictions (e.g., only files in `/tmp` are readable), but have not been successfully employed for quantitative restrictions because us-

[2] Or, in some cases (e.g., Janus [GWTB96]), to modify the request into a compliant form prior to allowing it to proceed.

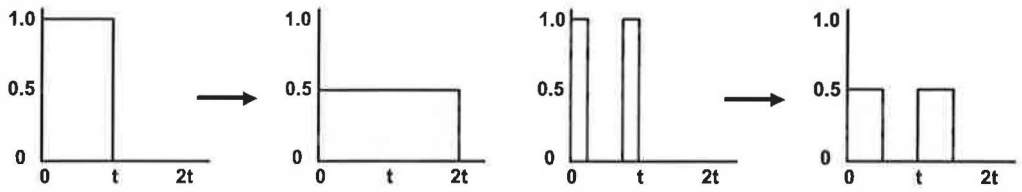


Figure 1. Desired effects on application execution time (x axis) under a resource-constrained sandbox that limits CPU share (y axis) to 50% when the application contains (a) no wait states, and (b) wait states. In the latter case, the sandbox should only cause the ready periods to get stretched out.

age of some resources (e.g., the CPU) does not require explicit application requests. In this paper, we extend these techniques to enforce quantitative restrictions over resource usage, with a scheme built upon core monitoring and control mechanisms that are a feature of most modern OSes.

3 Enforcing Quantitative Restrictions

Our strategy manages the allocation of system resources to an application by relying upon techniques for *instrumenting the application*, *monitoring its progress*, and as necessary, *controlling its progress* of execution. Progress metrics represent estimates of an application's resource consumption. Instrumenting (using tools such as [BG99, HB99]) allows us to inject code into the application and intercept its API calls on the fly. Since some system resources such as CPU and memory can be accessed without going through a high-level API call that can be intercepted, we control the resources available to the application both at the point of the request (e.g., when sending a message), and between resource requests (e.g., between memory allocations). These techniques leverage a core set of user-level mechanisms that are provided by most modern OSes such as priority-based process scheduling, page-based memory protection, and fine-grained timers.

In the rest of this section, we describe how this strategy can be used to control application consumption of three representative resources: CPU, memory, and network. The goal of controlling resource consumption can be twofold: to simply prevent an application from overusing system resources and starving other applications, or to provide a soft guarantee of and weighted fair sharing of resources to the controlled applications. The latter goal can create, for each application, a virtual execution environment that simulates a physical machine with the prescribed resource limitations. However, meeting this goal requires that extra resources cannot be given to the constrained application even if available. The techniques described in the rest of this paper address this more general goal.

3.1 CPU Resources

Here, the quantitative restriction is to ensure that the application receives a stable, predictable processor share. From the application's perspective, it should appear as if it were executing on a virtual processor of the equivalent speed.

Constraining CPU usage of an application utilizes the general strategy described earlier. The application is sandboxed using a monitor process that either starts the application or attaches to it at run time. The monitor process periodically samples the underlying performance monitoring infrastructure to estimate a progress metric. In this case, progress can be defined as the portion of its CPU requirement that has been satisfied over a period of time. This metric can be calculated as the ratio of the allocated CPU time to the total time this application has been ready for execution in this period. However, although most OSes provide the former information, they do not yield much information on the latter. This is because few OS monitoring infrastructures distinguish (in what gets recorded) between time periods where the process is waiting for a system event and where it is ready waiting for another process to yield the CPU. To model the virtual processor behavior of an application with wait times (see Figure 1 for a depiction of the desired behavior), we use a heuristic to estimate the total time the application is in a wait state. The heuristic periodically checks the process state, and assumes that the process has been in the same state for the entire time since the previous check.

The actual CPU share allocated to the application is controlled by periodically determining whether the granted CPU share exceeds or falls behind the desired threshold. The guiding principle is that if other applications take up excessive CPU at the expense of the sandboxed application, the monitor compensates by giving the application a higher share of the CPU than what has been requested. However, if the application's CPU usage exceeds the prescribed processor share, the monitor would reduce its CPU quantum for a while, until the average

utilization drops down to the requested level. While the application is waiting for a system event (e.g., arrival of a network message), it is waiting for resources other than the CPU. Consequently, the time in a waiting state is not included in estimating the CPU share and the application would not get compensated for being in a wait state. For this scheme to be effective, the lifetime of the application needs to be larger than the period between sampling points where the progress metric is recomputed.

3.2 Memory Resources

The quantitative restriction of interest here is the amount of physical memory an application can use. The sandbox would ensure that physical memory allocated to the application does not exceed a prescribed threshold. Monitoring the amount of physical memory allocated to an application is straightforward. The monitoring infrastructure on all modern OSes provides this information in the form of the process working set (resident set) size. The progress metric is the application's peak working set size over a period. No control is necessary when the progress is less than the threshold.

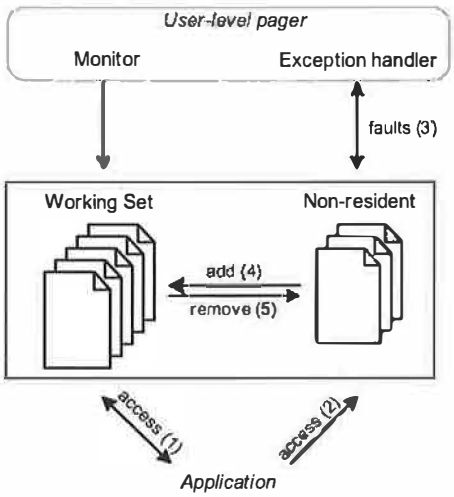


Figure 2. A user-level strategy for controlling application physical memory usage. The application has normal access to pages in its working set (1). When it accesses a non-resident page (2), a page fault is triggered (3). The user-level pager adds this page to the working set (4), and removes extra pages when working set size is above the threshold (5).

However, it is more involved to control the application behavior in case the OS allocates more physical pages than the threshold. The problem is that these resources are allocated implicitly subject to the OS memory management policies. The basic idea is to have the monitor act as a user-level pager on top of the OS-level pager, relying on an OS-specific protocol for voluntarily relinquishing the surplus physical memory pages allocated to the application (see Figure 2). Also, unlike the CPU case where periodic monitoring and control of application progress is required, here the monitoring and control can adapt itself to application behavior. The latter is required only if the application physical memory usage exceeds the prescribed threshold, which in turn can be detected by exploiting OS support for user-level protection fault handlers.

3.3 Network Resources

Here, the quantitative restriction refers to the sending or receiving bandwidth available to the application on its network connections. Unlike CPU and memory resources, application usage of network resources involves an explicit API request. This permits the monitoring code injected into the application to keep track of the progress (i.e., amount of data sent/received over a time window) and estimate the bandwidth available to the application. Control is straightforward: if the application is seen to exceed its bandwidth threshold, it can be made compliant by just stretching out the data transmission or reception over a longer time period (e.g., by using fine-grained sleeps). The amount of delay is calculated so that the bandwidth at the end-point is not above the prescribed threshold.[3]

3.4 Other issues

Integrated and implicit resource usage Applications do not access system resources in an isolated fashion. For instance, accessing a non-resident virtual memory page results in the triggering of an interrupt handler, transfer of a page from disk, accompanied with optional swapping out of a resident page and possible enlargement of the process working set size. To correctly handle such coupled accesses to system resources, we need to take into account effects such as increased CPU usage due to OS activity triggered on behalf of the application and additional disk usage because of reduced availability of physical memory pages.

Our sandboxing strategy factors in the above effects by appropriately defining the progress metric to reflect both *explicit* and *implicit* resource usage. The overall resource usage is forced to adhere to the requested limits by controlling the explicit requests. For example, even though an application's disk bandwidth usage due to paging is not controllable at the user level, its aggregate disk bandwidth usage can be reduced by controlling

[3] For clarity of description, we restrict our attention in this paper to synchronous communication operations and also assume that the data transmission rate in the network is not the bottleneck. The approach needs to be refined slightly to handle situations where communication operations are asynchronous.

explicit disk requests such as file read/write. As a last resort, quantitative rate-based limits on resource usage can be enforced by controlling allocation of CPU resources to the application.

Security concerns Given the user-level nature of our solution, a concern might be that an application can escape the sandboxing controls by bypassing our instrumented code. Currently we address this problem by having an *enforcer* process periodically verify that an application is adhering to its resource limits. The enforcer process terminates the offending process if it finds that the latter's resource consumption cannot be brought down below prescribed thresholds. As part of our future work, we are working on developing a finer granularity scheme that prevents code modification once sandboxing code is injected into the application, and additionally ensures at run time that the sandboxing code is not bypassed.

4 Implementation on Windows NT

This section discusses NT-specific issues and demonstrates the control of CPU, memory, and network resources with experiments. The implementation and performance results below refer to NT 4.0, service pack 5, running on Pentium II 450 MHz machines.

4.1 Constraining use of CPU resources

Monitoring progress The CPU monitor is attached as a callback routine of the fine-grained *multimedia timers*, and is triggered every 10ms with high accuracy using a technique introduced in [Gri98]. Note that the scheduling quantum on NT is at least 20ms for the workstation installation and 120ms for the server installation. The monitor obtains an application's CPU usage in terms of kernel time and user time through system API calls. The kernel time refers to the time the application is executing in kernel mode. However, this statistic does not account for all OS activity performed on behalf of the application. For instance, the overhead of memory paging is not included in per-process statistics, instead being recorded in the per-processor statistic. As a heuristic, we estimate the application's portion of this non-accounted kernel time by considering the ratio of the number of application events triggering such kernel activity (e.g., page faults) to the overall system-wide number of such events.

As described in Section 3, the monitor estimates process wait time within a time window by checking the process state and accumulating the time slots at which the process is found waiting. Although NT allows examining process state via its performance counter infrastructure, this incurs high overhead (on the order of milliseconds). Instead, we employ a heuristic that infers process state based on thread contexts. We observe that a thread can be in a wait state only when it is executing a function inside the kernel. Recognizing that if the thread is not blocked it is unlikely to stay at the same place in kernel code, the heuristic checks the instruction pointer register to see whether a trap instruction (int 2Eh) has just been executed, and whether any general registers have changed since the last check. If the same context is seen, it regards the thread as being in a wait state, with the process regarded as waiting if all of its threads are waiting.

Controlling progress Based on the progress metric, the controlling code decides whether or not to schedule the process in the next time slot. Although this decision could be implemented using OS support for suspending and resuming threads (which we use in our Linux implementation), the latter incurs high overheads. Consequently, we adopt a different strategy that relies on fine-grained adjustment of application process priorities to achieve the same result.

Our approach requires four priority classes (see Figure 3), two of which encode whether CPU resource are available or unavailable to the application. The monitor runs at the highest priority (level 4), and a special compute-bound "hog" process runs at priority level 2.[4] An application process not making sufficient progress is boosted to priority level 3, where it preempts the hog process and occupies the CPU. A process that has exceeded its share is lowered to priority level 1, allowing other processes (possibly running within their own sandboxes) or in their absence, the hog, to use the CPU. Note that this scheme allows multiple sandboxes to coexist on the same host.

Priority level	CPU available	CPU not available
4	`Monitor`	`Monitor`
3	**Application**	
2	`Hog`	`Hog`
1		**Application**

Figure 3. Controlling application CPU availability by changing process priorities.

Effectiveness of the sandbox Our experiments show that this implementation enables stable control of CPU resources in the 1% to 97% range. When the requested share is above 97%, the measured allocation includes perturbations from background load (the performance monitor, system processes, and the sandboxing code).

[4] Hog runs at a very low priority and executes only when no other normal applications are active, ensuring that even when CPU resources are available, each application does not receive more than its prescribed share.

The interference from sandboxing code consists of the monitor overhead and bursty allocation of resources to the hog process over long runs (this is an NT feature for avoiding starvation). The overhead of adjusting the priority is negligible. To measure the overall costs of running an application within a sandbox, we compare the wall-clock execution time of a synthetic CPU-intensive application running within and outside of a sandbox. On average, this application took 35.529 seconds to finish when running alone, and took 36.064 seconds when running inside a sandbox prescribing a CPU share of 100%, indicating an overhead of about 1.5%.

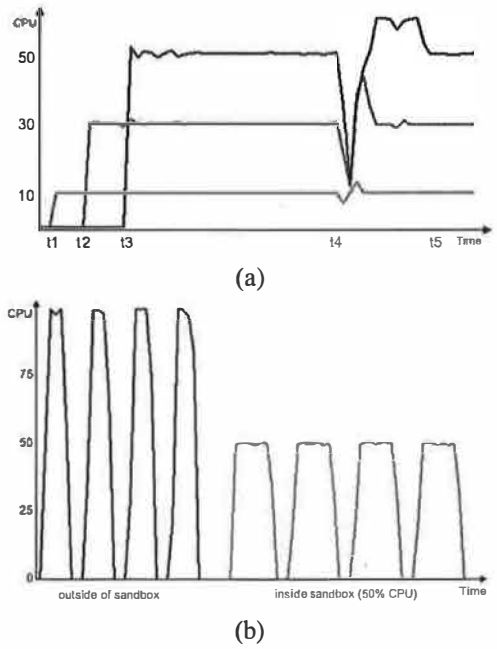


Figure 4. (a) Weighted CPU sharing for multiple applications. (b) Constraining CPU share for applications with wait states.

Figure 4(a) is a snapshot of the performance monitor display showing three sandboxed applications running on the same host. They start at times $t1$, $t2$, and $t3$, requesting 10%, 30%, and 50% of the CPU share, respectively. With the total CPU load at 90%, all three applications receive a steady CPU share until time $t4$, when we deliberately perturb the allocation by dragging a window. This causes the total available CPU to decrease drastically (because of the kernel activity), and a sharp decrease in the immediate CPU shares available to each application. However, this drop is compensated with additional CPU resources once the system reacquires CPU resources (end of window movement). These results indicate that the sandbox can support accurate and stable CPU sharing with resilient compensation.

Figure 4(b) shows the execution of an application that sleeps periodically, without sandboxing (left) and with a sandboxed CPU share of 50% (right). The working time with the sandbox is twice the amount on the left, corresponding to the halved CPU resource. More importantly, the sleep (wait) time is kept the same, consistent with Figure 1 and verifying the effectiveness of our state-checking heuristic.

4.2 Constraining use of memory resources

Monitoring progress An API call, `GetProcessMemoryInfo`, provides information about the resident memory of a process. Unlike the CPU case, the sampling of this information can be adapted to the rate at which the application consumes memory resources. To estimate the latter, we integrate the sampling with the controlling scheme described below.

Controlling progress As described in Section 3, controlling progress of memory resources requires the sandboxing code to relinquish surplus memory pages to the OS. To do this, we rely on a convention in NT: pages whose protection attributes are marked `NoAccess` are collected by the swapper.

The same core OS mechanism, user-level protection fault handlers, is used to decide both (a) *when* a page must be relinquished, and (b) *which* page this must be. Our scheme intercepts the memory allocation APIs (e.g., `VirtualAlloc` and `HeapAlloc`) to build up its own representation of the process working set. When the allocated pages exceed the desired working set size, the extra pages are marked `NoAccess`. When such a page is accessed, a protection fault is triggered: the sandbox catches this fault and changes page protection to `ReadWrite`. Note that this might enlarge the working set of the process, in which case a FIFO policy is used to evict a page from the (sandbox-maintained view of the) working set. The protection fault handler also provides a natural place for sampling the actual working set size, since a process's consumption of memory is reflected by the number of faults it incurs.

A few additional points need clarification. The implementation is simplified by not evicting pages containing executable code, so this limits the least amount of memory that can be constrained. Eviction at the sandbox level may or may not cause the page to be written to disk although these pages are excluded from the process working set; when the system has large amounts of free memory, NT maintains some pages in a transition state delaying writing them to disk. Note that with our design, if the application is running within its memory limits, it will not suffer from any runtime overhead (except that of intercepting API calls). Beyond that point, the overheads are a function of process virtual memory locality

behavior.

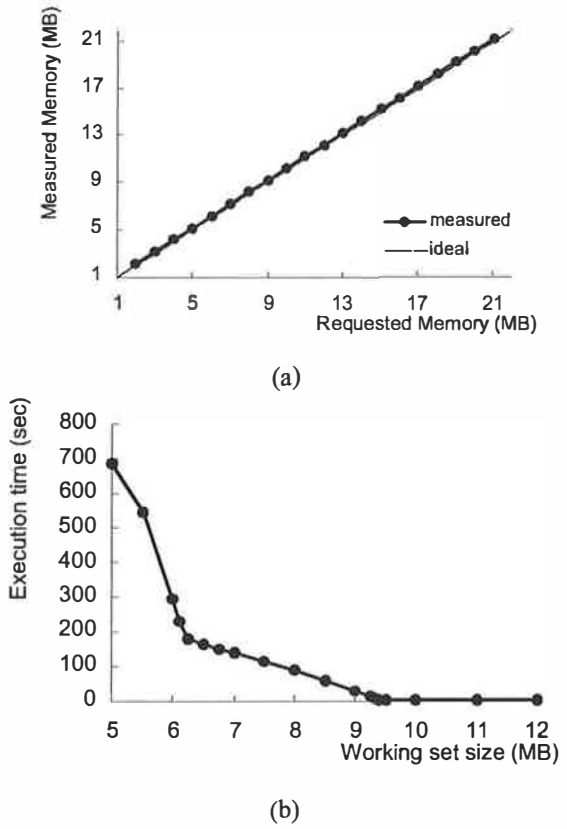


Figure 5. (a) Controlling the amount of physical memory utilized by an application. (b) Execution time as size of working set varies.

Effectiveness of the sandbox Our experiments show that, on a 450 MHz Pentium II machine with 128MB memory, this sandbox implementation can effectively control actual physical memory usage from 1.5MB up to around 100MB. The lower bound marks the minimal memory consumption when the application is loaded, including that by system DLLs.[5] The upper bound approximates the maximum amount of memory an application can normally use in our system. The memory overhead includes 64KB for the code injected into application address space and 4 bytes for keeping track of each page in the working set. The overhead of intercepting a memory allocation call is measured as 1.07μs when the specified memory constraints are above the working set size (thus no page fault is incurred). When the constraints are below the required working set size, process memory locality behavior determines the overhead. However, because of our CPU accounting scheme, only this process's execution time is affected.

[5]To compare, a "Hello, world" program consumes about 500KB memory and one that creates a TCP socket consumes 1MB memory.

Figure 5(a) shows the requested and measured physical memory allocations for an application that has an initial working set size of 1.5MB and allocates an additional 20MB of memory. The sandbox is configured to limit available memory to various sizes ranging from 2MB to 21MB. As the figure shows, the measured memory allocation of the application (read from the NT Performance Monitor) is virtually identical to what was requested.

Figure 5(b) demonstrates the impact of the memory sandbox on application execution time. The application under study has a memory access pattern that produces page faults linearly proportional to the non-resident portion of its data set. In this case, the application starts off with a working set size of 1.5MB and allocates an additional 8MB. The sandbox enforces physical memory constraints between 5MB and 12MB. As the figure shows, the execution time behavior of the application can be divided into three regions with different slopes. When the memory constraint is more than 9.5MB, all of the accessed data can be loaded into physical memory and there are no page faults. When the memory constraint is below 9.5MB, total execution time increases linearly as the non-resident size increases, until the constraints reaches 6.25MB. In this region, page faults occur as expected but the process pages are not written to disk. When available memory is below 6.25MB, we observe heavy disk activity. In this segment, the execution time also varies approximately linearly, with the slope determined by disk access characteristics. These experiments show that our sandboxing scheme does not produce any anomalous page faulting behavior.

4.3 Constraining available network bandwidth

Monitoring and controlling progress As described in Section 3, we intercept socket APIs (*accept*, *connect*, *send*, and *recv*) to monitor and control available network bandwidth.

Effectiveness of the sandbox The effectiveness of the sandbox is evaluated on a pair of Pentium II (450 MHz) machines connected to a 10/100 auto-sensing Fast Ethernet hub. The application consists of a server and one or more clients in a simple ping-pong communication pattern. The peak bandwidth measured without the sandbox is 9.7MBps, whereas the sandbox permits effective control of bandwidth over the range 1Bps to 8.8MBps. The overhead of sandboxing due to API interception lowers the maximum achievable bandwidth by about 0.4%, measured by comparing the case when the application is running alone and the case when it is running inside the sandbox with a bandwidth threshold that will never be reached.

Figure 6(a) shows the perceived bandwidth (at both

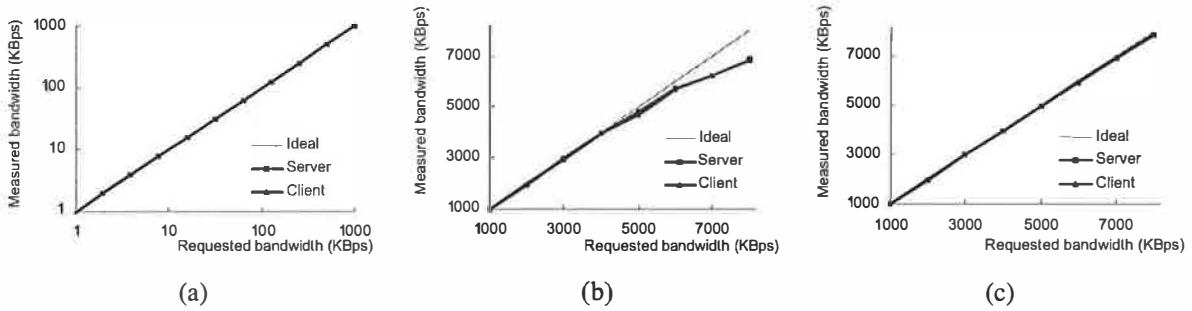


Figure 6. Measured bandwidth (a) for 1KB messages as requested bandwidth varies from 1KBps to 1000KBps, (b) for 1KB messages as requested bandwidth varies from 1000KBps to 8000KBps, (c) for 10KB messages as requested bandwidth varies from 1000KBps to 8000KBps.

server and client), measured by the application, as the sandbox constrains network bandwidth from 1KBps to 1000KBps. In the figure, the server and client bandwidth lines are virtually indistinguishable from each other and within 1% of the requested limit. Figure 6(b) shows the same measurement when bandwidth constraints are applied in the range 1000KBps to 8000KBps. When the requested bandwidth is below 4000KBps, the sandbox enforces the request with an error of at most 2%. However, error grows for higher bandwidths. For example, at 8000KBps, the sandbox can sustain a bandwidth at best 16% lower than requested. This is mainly due to the inaccuracy of the fine-grained sleep at sub-millisecond level on NT as well as the overhead of API interception. Figure 6(c), using 10KB messages, shows that the sandbox can be used to accurately sustain higher requested bandwidth as long as the message size scales proportionally. Here, the error between measured and requested bandwidths is less than 2%.

4.4 Experience with NT implementation

We have used the sandbox for various kinds of applications, including Microsoft Windows Media Player and Apple Quicktime Player for playing video streams in presence of compute and memory-intensive background processes. We can guarantee a smooth streaming video (without perceptible frame loss) either by sandboxing the background processes or by sandboxing the media players with a high CPU share. Our experiments have highlighted some limitations that we plan to address in future work. When we play two media players at the same time with one of them receiving a small CPU share, we observe frame loss in the other player due to priority inversion. The media player receiving the smaller share holds on to a shared resource (e.g., the sound card) even when it is not scheduled, causing the other player to drop frames. This problem could be addressed by intercepting the API calls used for acquiring such resources and integrating a priority inheritance mechanism into our policy for controlling CPU progress. In addition, our existing strategy cannot guarantee predictable resource allocation for hard real-time applications. This limitation can be fixed by using a real-time scheduling policy to determine allocation of CPU resources, instead of the current greedy policy.

4.5 Differences in Linux implementation

Linux provides support very similar to Windows NT for instrumenting application binaries, and monitoring and controlling resource consumption. For instance, library functions such as the sockets and memory allocation APIs, can be intercepted by preloading shared libraries. The mechanisms and performance of network bandwidth control are identical across the Windows NT and Linux platforms. However, there are small differences in how CPU and memory resources are constrained under Linux.

Controlling CPU resource Adjusting scheduling priorities requires superuser privilege on Linux. Therefore, we use a scheme based on thread suspend/resume: the sandbox sends an application a SIGSTOP signal to suspend and a SIGCONT signal to resume its execution.

Controlling memory resource Linux provides a `setrlimit` API for limiting the maximum amount of physical memory a process can use. However, current versions of the kernel (e.g., v2.2.12) do not enforce this constraint. Consequently, we adopt a scheme identical to the one on NT. However, unlike on NT, where an implicit protocol (using `NoAccess` protection bits) between the OS and the application permits the former to collect pages not required by the latter, no such protocol exists on Linux. The page protection bits can be set as on NT, but the kernel swapper (`kswapd`), does not check the page attributes to decide which page must be swapped out.

We get around this problem somewhat inelegantly by

handling the swapping ourselves. First, we intercept memory allocation functions (e.g., malloc) to make sure that only the requested amount of physical memory is kept valid; all other memory pages are protected to be unavailable for access. When a page fault happens due to invalid access, we pick another page (in FIFO order) from the resident set (maintained by the sandboxing code), save its contents to our own swap file, and take it out of the resident set using the `munmap` mechanism in Linux. Subsequently, an invalid access requires that the saved contents be mapped back to the corresponding virtual address.

5 Extensibility of User-level Sandboxing

This paper has described a general, user-level strategy for building sandbox environments that has been tested on both Windows NT and Linux. It is interesting to observe that modern OSes provide sufficient support to permit implementation of quantitative constraints at the user level. The shared library support enables interception of system APIs; the monitoring infrastructure makes possible acquiring of almost all necessary information; the priority-based scheduling, debugger processes, and signal handling mechanisms allow the adjustment of an application's CPU usage; the memory protection and memory-mapped file mechanisms permits control of the amount of physical memory available to an application. Finally, the socket interface gives direct control of network activities. Most resources in an operating system can benefit from some combination of these techniques.

In fact, user-level approaches provide more flexibility in deciding the granularity, the policies, and monitoring/controlling mechanisms available for enforcing sandbox constraints. We demonstrate this extensibility by customizing our process-level sandbox implementation on Windows NT to limiting resource usage at the level of thread and socket groups, instead of the default process granularity assumed in Section 3. The required modifications were simple, just involving changes in the progress expressions used in the monitoring code and some specialization of the controlling code.

Controlling CPU usage of thread groups Figure 7 shows a snapshot of the system CPU usage (as measured by the NT Performance Monitor) for an application with two groups of threads, each of which is constrained to a total CPU share of 40%. The application itself consists of three threads which start at different times. Initially, the first thread starts as a member of thread group one and takes up a 40% CPU share. The second thread starts after ten seconds and joins thread group two. It also gets 40% of the CPU share, the total capacity of this thread group. After another ten seconds, the third thread joins thread group two. The allocation for the second thread group adjusts: the third thread gets a 30% CPU share and the second thread receives a 10% CPU share, keeping the total CPU share of this thread group at 40%. Note that the CPU share of the first thread group is not affected, and that we are able to control CPU usage of thread groups as accurately as at the process level. Currently, the resource allocation to threads in the same group is arbitrary and not controlled. However, one could set up a general hierarchical resource sharing structure, attesting to the extensibility of the user-level solution.

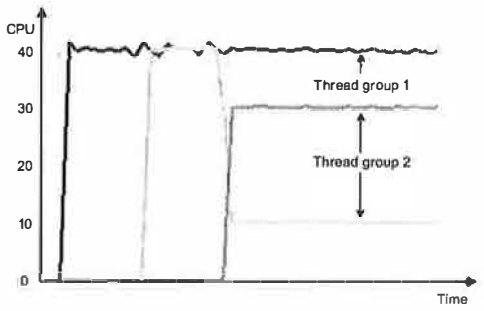
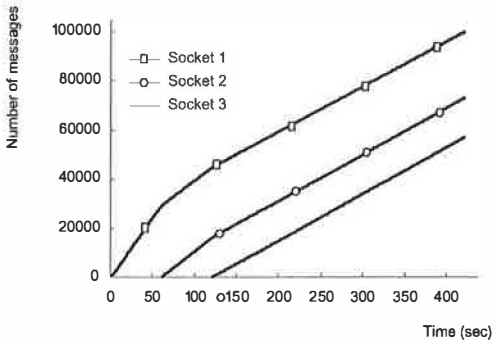


Figure 7. Control of CPU usage at the level of thread groups.

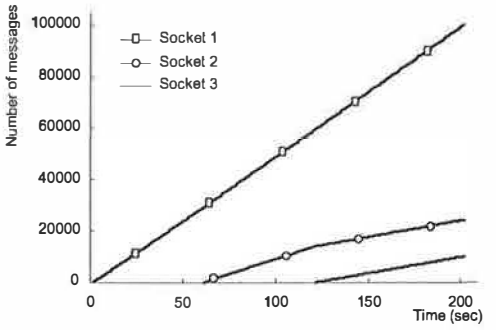
Controlling bandwidth of socket groups Figure 8 shows the effect of restricting network bandwidth at the level of socket groups, where the total bandwidth of a socket group is constrained. The application used in the experiment consists of one server instance and three client instances. The server spawns a new thread for each client, using a new socket (connection). The communication pattern is a simple ping-pong pattern between the clients and the server.

Figure 8(a) shows the performance of server threads when the bandwidth constraint is enforced at the process level. The total network bandwidth is restricted to 6MBps. The clients and server exchange 100,000 4KB-sized messages. The figure shows the number of messages sent by the server to each client as time progresses. The first client starts about the same time as the server and gets the total bandwidth of 6MBps (as indicated by the slope). The second client starts after one minute, sharing the same network constraint. Therefore, the bandwidth becomes 3MBps each. The communication is kept at this rate for another minute until the third client joins. This makes all three of them transmit at a lower rate (2MBps). As a result, the first client takes more than 400 seconds to complete its transmission, due to the interference from the other two clients.

Figure 8(b) shows the case where the first client needs



(a)



(b)

Figure 8. Control of network bandwidth (a) at process level, (b) at socket group level.

to receive a better and guaranteed level of service. Two socket groups are used, with the network bandwidth of the first constrained to 4MBps and that of the second group to 2MBps. Clients start at the same times as before. However, the performance of the first client is not influenced by the arrival of the other two clients. Only the two later clients share the same bandwidth constraint. In consequence, the first client takes only 200 seconds to finish its interactions.

These experiments demonstrate that user-level sandboxing techniques can be used to create flexible, application-specific predictable execution environments for application components of various granularity. As a large-scale application of such mechanisms, we have exploited these advantages in other work [CK00] to create a cluster-based testbed that can be used to model execution behavior of distributed applications under various scenarios of dynamic and heterogeneous resource availability.

6 Conclusion and Future Work

This paper describes the construction of a user-level resource-constrained sandbox, which exploits widely available OS features to impose quantitative restrictions on an application's resource usage. It evaluates a concrete implementation of the sandbox on Windows NT, using three representative resource types as examples: CPU, memory, and network. Our evaluation shows that the user-level sandboxing approach can achieve accurate quantitative restrictions on resource usage with minimal run-time overhead, and can be easily extended to support application-specific constraining policies.

In future work, we plan to develop a security architecture that ensures sandbox compliance from malicious applications at a finer granularity and address problems arising from priority inversion and the absence of real-time scheduling.

Acknowledgments

We would like to thank Zvi Kedem, who suggested using different priority levels to efficiently control a sandboxed process' CPU usage, and Anatoly Akkerman and Arash Baratloo for their help with implementing the sandbox on Linux. We also thank Lionell Griffith for giving us his implementation of fine-grained timers on Windows NT. This research was sponsored by the Defense Advanced Research Projects Agency under agreement numbers F30602-96-1-0320, F30602-99-1-0157, and N66001-00-1-8920; by the National Science Foundation under CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, SPAWAR SYSCEN, or the U.S. Government.

References

[BDM99] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[BG99] R. Balzer and N. Goldman. Mediating connectors. In *ICDCS Workshop on Electronic Commerce and Web-based Applications*, 1999.

[BGOS98] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation

domains. In *Proc. of USENIX 1998 Annual Technical Conference*, Jun. 1998.

[CK00] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *Ninth IEEE Intl. Symposium on High Performance Distributed Computing*, 2000.

[DKS89] A. Demers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. In *Proc. SIGCOMM '89 Symposium*, Sep. 1989.

[ET99] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.

[GGV96] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheuler for multimedia operating systems. In *Proc. of 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[Gri98] L. Griffith. Precision NT event timing. *Windows Developer's Journal*, Jul. 1998.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. of 6th USENIX Security Symposium*, Jul. 1996.

[HB99] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. of 3rd USENIX Windows NT Symposium*, Jul. 1999.

[JLDB95] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the Rialto operating system. In *Proc. of 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[JR99] M. Jones and J. Regehr. CPU reservations and time constraints: Implementation experience on windows NT. In *Proc. of 3rd USENIX Windows NT Symposium*, Jul. 1999.

[MST94] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. of IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[NL97] J. Nieh and M. Lam. The design, implementation & evaluation of SMART: A scheduler for multimedia applications. In *Proc. of 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[Ric99] J. Richter. Make your windows 2000 processes play nice together with job kernel object. In *Microsoft Systems Journal*, Mar. 1999.

[RJMO99] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proc. of SPIE/ACM Conference on Multimedia Computing and Networking)*, Jan. 1999.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, 1993.

[WW94] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of 1st Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[WW95] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Jun. 1995.

[Zha91] L. Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. In *Proc. ACM Trans. on Computer Systems*, May 1991.

WindowBox: A Simple Security Model for the Connected Desktop

Dirk Balfanz *
balfanz@cs.princeton.edu
Princeton University

Daniel R. Simon †
dansimon@microsoft.com
Microsoft Research

Abstract

Breaches in computer security do not just exploit bugs in applications; they are often also the result of mismanaged protection mechanisms. The tools available to protect sensitive resources and networks are tedious to use, non-intuitive, and often require expert knowledge. As a result, many PC and workstation users end up administering their system security poorly, creating serious security vulnerabilities. This paper presents a new security model, WindowBox, which presents the user with a model in which the workstation is divided into multiple desktops. Each desktop is sealed off from the others, giving users a means to confine the possibly dangerous results of their actions. We have implemented our security model on Windows 2000, leveraging the existing desktop metaphor, the ability to switch between multiple desktops, and specific kernel security mechanisms.

1 Introduction

Today's typical computing environment no longer consists of a centrally managed mainframe accessed through terminals; more often it consists of networked, Internet-aware PCs and workstations. The security threat model has thus changed in several ways:

- Because individual users have greater control over the PCs or workstations they typically use, they also have greater responsibility for the administration of these machines and their security. Where knowledgeable administrators once made professional decisions about security risks, those decisions (such as permission and denial of access to system and data resources) are increasingly in the hands of relatively uninformed users.
- The more open, distributed architecture of PCs and workstations makes it possible for users to introduce new applications and even operating system modifications to their machines – opening up another avenue of attack, through viruses, Trojan horses and other malicious applications.
- Internet connectivity has greatly expanded the range of possible attackers with access to a user's machine.

Various tools are already available to combat each of these threats. For example, existing discretionary access control mechanisms are usually available to limit access by (potentially hostile) applications to files. Firewalls and proxies can thwart certain kinds of network attacks. Finally, sandboxing has become a popular method for restricting a process' privileges to a subset of its owner's privileges, usually in the case where that process is executing some untrusted code. However, in practice we often see these techniques fail, not necessarily because they were poorly implemented, but because they are being poorly applied. The reason for this is that the currently used techniques, taken on their own, tend to be too complex for ordinary users to administer effectively. Taken together, they can be completely overwhelming even to fairly experienced users. As a result, confused users, faced with the responsibility of using and managing these techniques, often make ill-informed decisions with misunderstood implications – including, perhaps, serious security vulnerabilities.

It is thus imperative that user-administered security tools present the user with an easily comprehensible, intuitive model that allows users to understand the implications of their security policy decisions. For example, one idea they can understand is that of absolute physical separation. Given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines.

*Department of Computer Science, Princeton, NJ 08544
†Microsoft Research, One Microsoft Way, Redmond, WA 98052

In this paper, we describe a model, WindowBox, which is based on this idea of complete separation, and thus allows users to make reasonable security decisions with a clear understanding of their implications. We have implemented the model on a Windows 2000 workstation complete with a simple user interface that permits the user to manage the workstation's security in a natural, intuitive way.

In the next section, we will analyze existing security mechanisms and explain why they fail to deliver, especially in the hands of "average" users. In Section 3 we introduce the WindowBox model in more detail, before we explain our implementation on Windows 2000 in Section 4.

2 Existing Security Mechanisms

2.1 Access Control Lists

In the traditional, mainframe-based model, users wish to restrict access to their resources (which are almost always data files) to some limited set of users. Conceptually, an access matrix [3] describes what each user can do to each file. We often find that in a given implementation, the access matrix is replaced by a slightly less general, but more efficient mechanism. In Windows 2000, and other operating systems, Access Control Lists (ACLs) are used to restrict access of users to files. In practice, the vast majority of such restrictions limit access to a resource to a single owner (or not at all). In this manner files, for instance, can be fairly easily made "private" (accessible only to the file's owner) or "public" (accessible to all the system's users).

In the PC/workstation setting, however, a user may own many other objects – systems resources, communications resources, and so on. These are often more difficult to control by ACL, for various reasons: the object may perhaps be created and used by applications without the owner's knowledge in the first place; its ACL may not be accessible to the owner, for lack of an appropriate user interface; or it may be (as in the case of remotely accessible resources) difficult for the owner to tell who should or should not have access to the object – or even who is capable of attempting to access it.

Moreover, deciding on the appropriate ACL for every individual object (even whether it is "public" or "private") is a complicated, tedious and distracting task. Hence systems typically apply defaults to construct an ACL without consulting the user (such as when a new file is created). Since these default ACLs are assigned in the absence of information about the context in which they are created, they often conflict with the user's intentions.

2.2 Application-Level Security

When the resource in question is an application, it is typically the application itself that is made responsible for its own security. The operating system provides applications with various security services (such as authentication for remote connections); using these services, an application can limit access to itself as specified by the user running it, thus (hopefully) preventing its own exploitation by an external attacker.

This application-oriented approach has two major disadvantages:

- The user making the access decisions must deal with a different configuration procedure for each application – hopefully involving the same system-provided infrastructure, but sometimes even depending on application-specific authentication and access control mechanisms.
- Each application's security is implemented separately – and therefore has its own independently buggy security. A user would need to know all the security holes in each application, and how to defend against attacks on each one, in order to secure his or her system against attack.

A preferable alternative would be for the user to be able to set access control rights for any application from "above", requiring any inter-process communication reaching the application to be authenticated as coming from a permitted user or user/application pair. A natural analogy is the file system: just as access (of any sort) to a file can be limited to a particular set of users, so too should access to an application. (Indeed, as the distinction between applications and data blurs, so does the distinction between the two types of access.)

2.3 Firewalls and Proxies

The problem of controlling access to an application becomes even more difficult when the application is intended to be "network-aware". A well-connected PC or

workstation, for example, runs a number of applications of various types; these may include both traditionally server-based applications (HTTP servers, FTP servers and the like) and traditionally user-oriented (but increasingly network-aware) applications. All of these may at various times send data onto the network or monitor and read arriving network data, using network services provided by the operating system. We might assume that the careful user avoids installing deliberately malicious applications; however, even established commercial applications can have security holes which may be exploited by hackers sending them unexpected data.

To protect against the possible existence of such holes, many administrators of large networks of PCs/workstations install firewalls or proxies, which filter out network traffic that does not conform to the formats that the applications are expected to handle correctly. But applications may still have bugs which allow an attacker to subvert them by sending them data that is allowed through by the firewall or proxy,b ut has unanticipated effects due to bugs or design oversights in the application.

Therefore, the problem would be much more effectively addressed by enforcement of the following two restrictions:

- Limiting applications' access to the network to authenticated connections, with access control applied to these connections "from the outside," as discussed above.
- User-imposed blocking of applications involved in network communications (especially unauthenticated connections) from accessing sensitiveresources on the same machine.

For example, a personal HTTP server might be allowed to accept unauthenticated connections with the outside world, but be forbidden to write to files or communicate with other applications beyond an "internal firewall." The other side of this firewall might contain "private" applications permitted to communicate with each other, but not with the HTTP server,and not with the network at all except over authenticated connections to members of a particular group of trusted users (say, an Intranet).

2.4 Sandboxing

Meanwhile, the recent proliferation of mobile code via the Internet has spurred demand for mechanisms by which such mobile applications can be "sandboxed" (granted only restricted access to system resources, possibly including files, communication channels, miscellaneous system services and other applications). Originally, mobile code consisted in practice only of small "applets" with very limited functionality (and thus easy to sandbox tightly). Increasingly, however, the crisp lines between simple, casually distributed applets and "store-bought" applications, and between entire applications and individual components, have begun to blur. Sophisticated software is now being distributed over the Internet in the form of applets or even entire applications, and components distributed in this fashion are being used together in complicated ways. Even operating system updates, for instance, are being distributed over the Internet. The distinction between "trusted" and "untrusted" software is thus increasingly a continuum, with no software enjoying the confidence once (possibly too naively) accorded store-bought, "shrink-wrapped" software. It follows that some degree of sandboxing will become desirable for a wide range of applications and components.

A prominent example of the necessity of ubiquitous sandboxing is the use of cryptographic keys for sensitive functions such as electronic commerce or Intranet authentication. The susceptibility of such keys to compromise by viruses or security holes, and the dire consequences of such compromises, make it imperative that keys be accessed only by applications that can be trusted to use them properly (requesting direct user approval, for instance). A broadly enforced sandboxing policy could make such protection possible.

The major existing sandboxing framework is the Java virtual machine (VM), which can be used to allow "applets" (typically embedded in Web pages) restricted access to resources. This VM is only available for applications distributed in the form of a particular Java byte code, and then translated (with some performance overhead) into native machine code. The Java security model allows applications to be accompanied by digitally signed requests for various system access privileges; the user or administrator can decide whether to grant applets their requested privileges based on the identities of the digital signers.

This model ultimately requires the user to make difficult qualitative trust judgments, such as which permissions are too sensitive to trust to which applications. The model can thus lead to a proliferation of too many identities and permissions for the user to keep track of. At its worst, it simply adds an extra layer of complication to the ACL model: resources must be access-controlled

based not only on user identity, but also application origin. While it may be a useful tool for implementing a higher-level model, it is still far too complex for direct manipulation by ordinary users when applied to a great many applications.

3 A Model For Sandboxing-Based Security

The above issues suggest the following goals for a security model:

- It should unify the per-user access control functions of the ACL model with the security properties of firewall- and sandboxing-based models, so that users only have to deal with a single model;
- It should allow basic application-level security (restricting both access to the application and its range of permitted behavior) to be applied independent of the application;
- It should be simple enough to be managed via a natural, intuitive user interface.

The last goal is perhaps the most crucial, as well as the most difficult. Creating an understandable user interface for a security model is generally a daunting task, given the complexity and criticality of security administration. The only hope is to design a simple and natural model that users can grasp and manipulate intuitively, then present a UI which reflects this model. Otherwise, confused users will inevitably make dangerously mistaken security decisions – such as disabling all security features entirely, to avoid the inconvenience of dealing with them.

3.1 The Multi-Desktop Premise

We propose here a model based on an extremely simple idea: while users cannot really intuit the complex rules associated with zones or ACLs, one idea they can understand is that of absolute physical separation. Given a set of group or zone permission rules, a user will have a difficult time determining if it expresses his or her idea of security. But given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines. It may be that many users (particularly small businesses) are using multiple machines in this manner today to secure their sensitive data and applications from the Internet. And of course, large enterprises use proxies all the time to protect their internal machines from the world outside.

Moreover, users' business (both information and – to a lesser extent – applications) tends to divide up fairly cleanly into categories, such as "personal finance," "business/office/intranet," "Internet gaming," and so on. The amount of information flowing between these categories is typically relatively small, and therefore should be manageable through direct user intervention (carrying floppy disks between machines, or its drag-and-drop analog in the virtual context) without excessive strain on the user.

3.2 The WindowBox Security Model

In the WindowBox security model, the user can construct multiple desktops, which are kept completely isolated, except by explicit user action (such as a direct point-and-click command or response to a dialog or warning box). To a first approximation, the desktops start off completely identical, and are provided just so that the user can use different desktops for different tasks. As the user uses the different desktops, each one accumulates its own files, and applications in one desktop cannot access files in other desktops except by the aforementioned user action. To aid the user in consistently using the desktops for their specific purpose, each desktop has its own network access restrictions and code-verifying criteria (although the user would manipulate these only indirectly, by defining and configuring desktops). Many special-purpose applications would be confined to a single desktop; other, more general applications would be "installed" in multiple desktops, but would have different access rights, and possibly even different behaviors, in each desktop (for example, a word processor might have different defaults depending on whether it is being run in an enterprise/Intranet desktop or a personal one).

In some ways, the multiple desktops can be considered as representing different users logged on simultaneously. A key difference, however, is that simple user-mediated actions would always be able to transfer data between one desktop and another, and (if necessary) create new desktops or change the properties of existing ones. From the network's perspective, on the other hand, these desk-

tops would have very different security properties. For example, access to the private key necessary for authenticating as the user in a secure connection to a particular server may be restricted to applications in a particular desktop. Hence a server that only accepts secure connections would implicitly require that the user access it from only that desktop.

3.3 Examples

Most of the work of defining and configuring desktops should be a matter of choosing among standard desktop types with preset, mildly customizable attributes. We suggest a few natural ones here.

3.3.1 The Personal Desktop

A simple example of a useful separate desktop is a "personal" desktop to isolate sensitive personal (e.g., financial) applications and data from the rest of the user's machine. Applications in such a desktop would be limited to those handling such personal matters, plus a few trusted standard ones such as basic word processing. These applications would also be isolated from all interprocess communication with applications on other desktops, and files created by them would be inaccessible from any other desktop. Network access in this desktop would be limited to secure, authenticated connections with a small number of trusted parties, such as the user's bank(s) and broker(s); no general browsing or Internet connections would be permitted. Similarly, the authentication credentials required to establish authenticated connections to these trusted parties would be isolated in this desktop. A user with such a separate desktop should feel comfortable using the same machine for other purposes without fear of exposing sensitive personal data or functionality to attackers.

3.3.2 The Enterprise Desktop

Like personal data, a user's work-related data and applications are best kept isolated from the rest of the user's machine. In an "enterprise" desktop, only enterprise-approved work-related applications would be allowed to run, and network access would be limited to secure authenticated connections to the organizational network or intranet (and hence to the rest of the Internet only through the enterprise proxy/firewall). Again, all applications on this desktop would be "sandboxed" together, and denied inter- process communication with applications outside the desktop. The capability to authenticate to the enterprise network would also be isolated in this desktop. Such isolation would allow a user to access an enterprise Intranet safely from the same machine used for other, less safe activities.

Note that enterprise-based client-server applications actually benefit enormously from such isolation, because they typically allow the user's client machine to act in the user'sname for server access. Thus if an insufficiently isolated client application opens a security hole in the client machine, it may implicitly open a hole in the server's security, by allowing unauthorized attackers access to the server as if they were at the same authorization level as the attacked client. On the other hand, if the client application and all associated access rights are isolated in an "enterprise desktop," then malicious or vulnerable applications introduced onto the client machine for purposes unrelated to the enterprise are no threat to the enterprise server's security.

3.3.3 The "Play" Desktop

For games, testing of untrusted applications, and other risky activities, a separate desktop should be available with full Internet access but absolutely no contact with the rest of the machine. There may be multiple play desktops; for example, a Java-like sandbox for untrusted network-based applets would look a lot like an instance of a play desktop.

3.3.4 The Personal Communication Desktop

Since users are accustomed to dealing with email, Web browsing, telephony/conferencing and other forms of personal communication in an integrated way (as opposed to, for instance, receiving email in different desktops), these functions are best protected by collecting them in a single separate desktop. This desktop should run only trusted communications applications; those communications (email, Web pages, and so on) which contain executable code (or data associated with non-communications applications) would have to be explicitly moved into some other desktop to be run or used. For example, a financial data file contained in an email message from the user's bank would have to be moved into the personal desktop before being opened by the appropriate financial application. Note that some communications functions could also be performed in other

desktops; for example, a Web browser in the enterprise desktop might be used to browse the enterprise Intranet (to which applications – including browsers – in other desktops would have no access).

3.4 How WindowBox Protects the User

Before we look at our implementation of the WindowBox model, let us recap how WindowBox can prevent common security disasters. Consider, for example, users who like to download games from questionable Web sites. Every now and then, one of the downloaded games may contain a virus, which can destroy valuable data on people's machines. If the game is a Trojan Horse, it might also inconspicuously try to access confidential files on the PC and send them out to the Internet. If the users were employing WindowBox, they would download the games into a special desktop, from which potential viruses could not spread to other desktops. Likewise, a downloaded Trojan Horse would not be able to access data in another desktop.

As a second example, let us now consider the recent spread of worms contained in email attachments. For example, the Melissa worm (often called the "Melissa virus"), resends itself to email addresses found in the user's address book. On a WindowBox-equipped system, users would open email attachments in a desktop that is different from the desktop in which the email application is installed. This might be because the attachments logically belong in a different desktop, or simply because the user judges them not trustworthy enough for the email desktop. From that other desktop, the worm cannot access the email application, or the network, to spread itself to other hosts.

4 Implementation

We implemented the WindowBox security architecture on a beta version of Windows 2000 (formerly known as Windows NT 5.0 Workstation). In our implementation, the various desktops are presented to the user very much in the manner of standard "virtual desktop" tools: at any given time, the user interacts with exactly one desktop (although applications running on other desktops keep running in the background). There are GUI elements that allow the user to switch between desktops. When the user decides to switch to a different desktop, all application windows belonging to the current desktop are removed from the screen, and the windows of applications running in the new desktop are displayed. Windows 2000 already has built-in support for multiple desktops. For example, if a user currently works in desktop A, and an application in desktop B pops up a dialog box, that dialog box will not be shown to the user until he or she switches to desktop B. Windows 2000 provides an API to launch processes in different desktops and to switch between them. We simply had to provide GUI elements to make that functionality available to the user.

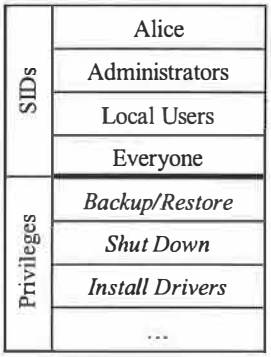


Figure 1: A sample access token

However, the desktops provided by Windows 2000 do not, in any way, provide security mechanisms in the sense of the WindowBox security architecture. Our implementation therefore had to extend beyond what is offered in Windows 2000. In this section we describe these extensions.

Before explaining how we represent desktops as user groups, and what changes we made to the NT kernel to implement WindowBox security, we will briefly recap the Windows NT security architecture.

4.1 The Windows NT Security Architecture

In Windows NT, every process has a so-called access token. An access token contains security information for a process. It includes identifiers of the person who owns the process, and of all user groups that person is a member of. It further includes a list of all privileges that the process has. Let's assume that Alice is logged on to her Windows NT workstation and has just launched a process. Figure 1 shows what the access token of such a process might look like: It includes an identifier (also called Security Identifier, or SID) of Alice as well as of all the groups she is a member of. These include groups

1.*Allow* Write Administrators
2.*Deny* Read Alice
3.*Allow* Read Everyone

Figure 2: A sample DACL

that she has explicitly made herself a member of, such as "Administrators," as well as groups that she implicitly is a member of (like "Everyone"). Since she is an administrator on her workstation, her processes get a set of powerful privileges (these privileges are associated with the user group "Administrators"). The figure shows a few examples: The "Backup/Restore" privilege allows this process to read any file in the file system, regardless of the file's access permissions. The "shut down" privilege allows this process to power down the computer, etc.

Access tokens are tagged onto processes by the NT kernel and cannot be modified by user-level processes[1]. When a user logs on to the system, an access token describing that user's security information is created and tagged onto a shell process (usually Windows Explorer). From then on, access tokens are inherited from parent to child process.

The second fundamental data structure in the Windows NT security architecture is the so-called security descriptor. A security descriptor is tagged onto every securable object, i.e. to every object that would like to restrict access to itself. Examples of securable objects include files or communication endpoints. A security descriptor contains, among other things, the SID of the object'so wner and an access control list. The access control list (also called Discretionary Access Control List, or DACL) specifies which individual (or group) has what access rights to the object at hand. It is matched against the access token of every process that tries to access the object. For example, consider a file with the access control list shown in Figure 2. What happens when Alice tries to access this file? Since her access token includes the Administrators group, she will have write access granted. However, the DACL of this file explicitly denies read access for Alice (let's forget for a moment that Alice's access token also contains the Restore/Backup privilege, which enables her processes to override this decision). Because the order of the DACL entries matters, it is not enough that the group "Everyone" (of which Alice is a member, as her access token specifies) has read access. The entry that denies Alice read access comes first, effectively allowing everyone but Alice read access. We can see that the combination of access tokens and security descriptors provides for an expressive and powerful mechanism to specify a variety of access policies.

[1] This is an oversimplification. In reality, there are some limited operations that a user-level process can do to an access token: It can switch privileges on and off and even (temporarily) change the access token of a process (for example, when a server would like to impersonate the client calling it). However, a process can never, of its own volition, gain more access rights than were originally assigned to it by the kernel.

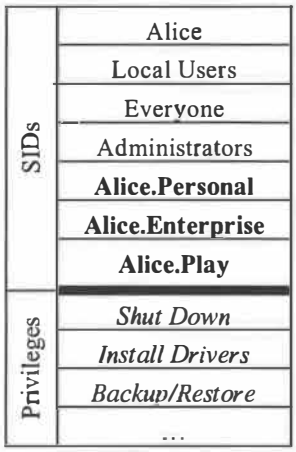


Figure 3: A sample access token with desktop SIDs

Only the kernel can modify the security descriptor of an object, and it will only do so if the process that is requesting modifications belongs to the owner of that object. Also, the access check described above happens inside the kernel. No user process can, for example, go ahead and read a file if the file's DACL forbids this.

4.2 Desktops as User Groups

Apart from the graphical representation to the user, we internally represent each desktop as a user group. For example, if Alice wanted three desktops for her home, work, and leisure activities, she could create three user groups called Alice.Personal, Alice.Enterprise, and Alice.Play. She would make herself a member of all three groups[2]. Now, whenever she logs on to her computer, her access token would look like the one shown in Figure 3. Note that the SIDs that represent her desktops are added to the access token. This happens automatically since Alice is a member of all these groups. We call these SID's desktop SIDs. They are marked as desk-

[2] In our prototype, this process is automated and happens when a new desktop is created.

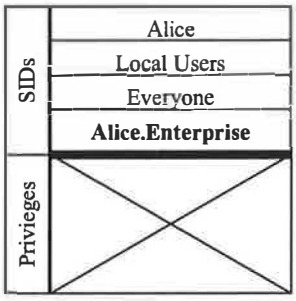


Figure 4: A sample access token of a desktop process

top SIDs in the access token (denoted by bold typeface in the picture), but are otherwise just normal group SIDs.

In our implementation, we create a desktop for every desktop SID in the access token when the user logs on. Continuing our example, when Alice logs on, we create three desktops. In every desktop, we start a shell (Windows Explorer)[3]. However, we limit what each shell can do by giving it a restricted token (the restricted token API has been introduced in Windows 2000 and allows processes to limit their own, or their children's, privileges): First, we remove all privileges from the access token. Then we remove all desktop SIDs except the one representing the desktop for which we are preparing the access token[4]. Lastly, we remove the "Administrators" SID from the token in a move to restrict access to system files, which usually allow write access to the Administrators group. Each shell is launched with this restricted token. Figure 4 shows the access token of the shell running in Alice's enterprise desktop.

When Alice now starts applications or processes in one of her desktops, they will inherit the restricted token of the desktop'ssh ell. Note that this also holds for ActiveX components and other executable content downloaded from the network, which runs inside descendants of the desktop's shell.

With the system described so far, we could already implement some kind of WindowBox security. If Alice judiciously restricted access to some files to the user group Alice.Enterprise (and only to that group), these files could only be accessed by applications running in the enterprise desktop. However, one of the major goals of the WindowBox security architecture is to relieve the user from the burden of making difficult access control decisions, and to "automate" this process. Furthermore, while the user can modify the DACLs of files, this is not true for all objects. For example, processes in one desktop can still communicate with processes in another desktop, and there is nothing that the user can do about this. For this reason, we introduced the concept of "confined" objects.

4.3 Confined Objects

A confined object is an object that belongs to a certain desktop. The idea is that an object confined in desktop A should not be accessible by any process from desktop B. We confine objects by tagging their security descriptor with the SID of the desktop they should be confined to. For example, to confine a file to Alice's enterprise desktop, we would add the SID Alice.Enterprise to the file's security descriptor. This extension of the security descriptor is our first modification to the NT kernel.

Our second modification makes sure that objects automatically become confined: Whenever the kernel creates a securable object (i.e., an object with a security descriptor) such as a file or a communications endpoint on behalf of a process, we confine that object to the desktop that the creating process runs in. Note that not all objects have to be confined. There are processes on the system (for example system services) that do not belong to any desktop because they are not descendants of any of the desktops' shells. The kernel can also create objects on its own behalf.

Our third modification concerns the access check performed in the kernel. The original access check implements the semantics of the access control list of an object with respect to the accessing process' access token as explained above. We changed the access check as follows:

1. Is the object confined? If so, go to 2; otherwise go to 3.
2. Check whether the process' access token contains a SID that is equal to the desktop SID the object is confined to. If so, go to 3; otherwise deny access.
3. Perform a normal access check.

[3] For the record we should mention that there will also be a fourth desktop that serves as a "root" desktop in which all applications have full privileges and access to the system. Alice should stay away from that desktop for her day-to-day work, but can use it for administrative tasks.

[4] The diligent reader will object that removing a group SID from an access token doesn't necessarily restrict the process' rights. Windows 2000 does the right thing: The SID is not completely removed, it is "disabled for positive DACL entries": If you disable the SID "Group A" in an access token, and then try to open a file that allows only access to Group A, this access will not be allowed. However, if you try to access a file that explicitly denies access to Group A, access *will* be denied.

Our modifications implement the WindowBox security architecture: The desktops are completely sealed off from each other. For example, a file saved by a word processor in one desktop will be confined to that desktop and cannot be accessed by any process from another desktop, including potentially malicious applications downloaded from the outside world. This goes beyond what default ACLs in Windows 2000 or the umask feature in UNIX offer: First, the confinement cannot be undone by an ordinarily privileged process (see below). Second, there is no customizable setting that the user has to decide on (i.e., what should the default ACL or umask be?). Object confinement is mandatory, and cannot be customized, or mismanaged, by the user. Also note that confined files are usually visible to other desktops, they are merely inaccessible. An access to a file confined to another desktop would fail in the same way that access to file belonging to another user would. However, by the same token, this also implies that files within a directory in another desktop will *not* be visible.

Notice that processes also become confined as they are created (they, too, are securable objects).

Once an object is confined, it takes a special privilege to "un-confine" it (by removing the confining SID from its security descriptor) or to confine it to another desktop. Since we strip all privileges from the shells (and their children) in each desktop, no application in any desktop can move objects from one desktop to another. However, in each desktop we provide one process that is not restricted in the same way as the shell and its children are. This process provides the GUI to switch to other desktops (the restricted processes in the desktops would not even have enough privileges to make that switch). That privileged process also serves as a COM server exporting the service to move objects between desktops. Every process can connect to that server and ask it to move an object from one desktop to another. The server can then decide whether or not to do so. In our implementation, it asks the user for confirmation before any object (such as a file) is moved between desktops.

4.4 Restricting Network Access

The system, as described so far, can already safeguard against a number of attacks if used consistently. Our user Alice should never do her finances in her play desktop, for example. Nor should she visit untrusted Web sites while she is in her enterprise desktop. To encourage her to abide by the latter rule, we have restricted network access for processes running in desktops.

We modified the kernel to deny any network access to a process running in a desktop (i.e., a process with a desktop SID in its access token). However, we also added a layer in the network stack that relays network calls of a desktop process (which would fail in the kernel) to the privileged COM server mentioned above. This privileged process can connect to the network, but will only do so if the requested network address satisfies the desktop's policy (e.g., it is explicitly included in a list of permitted addresses). If so, it connects to the requested network address and returns the handle representing the network connection back into the unprivileged process.

In our implementation, users can specify a different network access policy for each desktop. The mechanism used is powerful enough to express policies like: "only allow connections to the corporate intranet," "only allow connections to www.mybank.com," "deny any network access," etc.

4.5 Security Analysis

How secure is WindowBox? This question does not have a generic answer. Rather, we need to ask how secure a specific implementation of the WindowBox security model is. A "secure" WindowBox implementation does not allow malicious code to affect another desktop. For example, a virus should not be able to infect files in a different desktop, a data-gathering Trojan Horse should not be able to read files in other desktops, etc. In other words, there should be no *channels* between desktops, which malicious code could exploit. The work on covert channels has shown that it is not possible to close every covert channel. For the purposes of this paper, a covert channel can be defined as a means by which information could leak from one desktop to another. Note that this is considerably less powerful a channel than one which a virus could actually exploit to propagate itself to a different desktop. For that purpose, a virus would have to be able to write a file in one desktop, and then cause that file to be executed in a different desktop. While our situation is not as hopeless as with covert channels, we still believe that it is impossible to get formal assurance that no dangerous channels exist. Windows 2000 is too complex a system for us to hope to model it in a way that would yield relevant statements about a given WindowBox implementation.

What, then, can we say about the security of our WindowBox implementation? We tried to make our implementation as secure as possible by implementing it at as "low" a level as possible inside the kernel. The rationale

is that every malicious program has to go through certain parts of the kernel - most notably, the access control reference monitor - in order to do anything useful, including any attempt to access another desktop. Therefore, we placed the WindowBox enforcement code inside that reference monitor.

There are, however, covert channels left between desktops in our current implementation. For example, files that are not confined to a desktop could potentially be writable by one desktop and then readable by another desktop. To prevent this covert channel from turning into a channel that a virus could exploit, we made sure that none of the executable system files are writable from any desktop. Ultimately, we believe that a lot of scrutiny will be necessary to find and deal with other potential channels, and that for this reason a production version of WindowBox would need extensive testing and prolonged exposure to the security community.

The last part of the answer is that a WindowBox implementation would be most secure on top of an operating system that was designed in anticipation of this kind of security model. For example, we mentioned above that certain applications should be installed in only one desktop, or that applications should be installed separately in multiple desktops. Windows 2000 does not really allow us to do that. For example, applications written for Windows 2000 like to keep configuration data in the (system-wide) registry or use well-known (system-global) files to store information. Ideally, the operating system should be designed from the ground up with the WindowBox model in mind, thus eliminating potential cross-desktop channels that malicious programs could exploit. However, a few key modifications to Windows 2000 would go a long way towards supporting WindowBox more securely; for example, some parts of the registry could be replicated, with separate copies for each desktop, to allow applications to install transparently on some desktops but not others.

Finally,we would like to remind the reader that no system is more secure than the decisions of its user or administrator,and that defining a security model in which the protection of sensitive data is relatively convenient (as is the case in WindowBox) creates the possibility of implementations converging towards security that is not only free of serious intrinsic holes, but also usable enough to avoid many of the types of holes introduced by the unsafe practices of users battling cumbersome systems.

5 Related Work

We are not the first to recognize the specific security requirements of a ubiquitously networked world, especially in the light of mobile code. A standard goal is to prevent mobile code, or compromised network applications, from penetrating the system. The generic term for ways of achieving this goal is "sandboxing." The term was first used in [5] to describe a system that used software fault isolation to protect system (trusted) software components from potentially faulty (untrusted) software components. Perhaps the best-known example of sandboxing is the Java Virtual Machine. It interprets programs written in a special language (Java bytecode). It can limit what each program can do based on who has digitally signed the program and a policy specified by the local user. The biggest drawback of the Java approach is that it can only sandbox programs written in Java.

Our system sandboxes processes regardless of which language they have been written in. That, too, is not new. In [2], Goldberg et al. show how any Web browser helper application can be sandboxed. Their work, however, only targets processes that are directly exposed to downloaded content, and requires expertise in writing and/or configuring security modules.

Our system has certain resemblance to role-based access control in that one could think of our desktop SIDs as a user's different roles. In fact, in [4], Sandhu et al. imagine a system in which "a user might have multiple sessions open simultaneously, each in a different window on a workstation screen." In their terminology, each "session" comprises a certain subset of the user's roles. Hence, in each window the user would have a different set of permissions.

Another concept with similarities to our desktops is that of "compartments" in Compartmented Mode Workstations (CMWs). CMWs are implementations of the Bell-LaPadula model [1] found in high-security military or government systems. Like CMW, we introduce "mandatory" security to an otherwise "discretionary" security model. In [8], Zhong explains how vulnerable network applications – such as a Web server – can be made less of a threat to the rest of the system if they are run in special compartments, shielding the rest of the system. The WindowBox security model is in some sense weaker than the Bell-LaPadula model. We are merely trying to assist the user in separating his or her different roles. For example, covert channels from one desktop to another are much less of a concern to us than they are for

a Compartmented Mode Workstation: In CMWs, an application voluntarily surrendering its data is a problem, in WindowBox it is not (it is simply not part of the threat model).

Domain and Type Enforcement (DTE) can also be used to sandbox processes in a way similar to ours. In [6], Walker et al. explain how they limit what compromised network applications can do by putting them in "domains" that don't have write access to certain "types" of objects, for example system files.

What distinguishes our work from the long list of other sandboxing approaches is that all of the above use sandboxing as a flexible, configurable technique to enforce a given – usually complicated – security policy. We argue that figuring out a complex, customized security policy is too difficult a task for most users to handle. In contrast, in our system we do not use a general sandboxing mechanism to implement specific security policies. In fact, we don't have any security policy in the traditional sense, except for the requirement of strict separation of desktops. It is this simplicity of the model that we think shows great promise for personal computer security.

Another area of related research is that of user interfaces and security. Recently, more and more people have realized that poor user interface design can seriously jeopardize security (see, for example, [7]). We very much concur with the thrust of that research. Our approach, though, is more radical. Instead of suggesting better user interfaces for existing security tools, we propose a completely redefined security model. One of the features of WindowBox is that is naturally lends itself to a more intuitive user interface for security management. Users have to understand, and learn tools that visualize, only *one* concept - the fact that separated desktops can confine the potentially harmful actions of code.

6 Conclusions

In this paper, we argue that existing security mechanisms – while possibly adequate in theory – fail in practice because they are too difficult to administer, especially for a networked personal computer or workstation under the control of a non-expert user. We present an alternative to this dilemma, WindowBox, a security model based on the concept of complete separation. While it has similarities to some existing security mechanisms, it is unique in that we do not try to provide a general mechanism to enforce all sorts of security policies. In contrast, we have only one policy – that of complete separation of desktops. We believe that this "policy" is easy to understand and has promise for the connected desktop.

We found Windows 2000 to be a good platform to implement WindowBox on: We could leverage the existing desktop API and were able to restrict changes to the NT kernel to a minimum.

The WindowBox model and prototype raise several interesting questions: What kinds of hidden security vulnerabilities might they contain, and how might they be eliminated? For instance, instead of using our own user interface, should we haveus ed Window 2000's "Secure Attention Sequence"[5] to make sure that an application cannot trick the users into believing they are in a different desktop than they really are? How usable is a multiple-desktop environment for average users? (We conjecture that users' typical activities divide themselves up naturally in ways that correspond well with distinct desktops, but we have done no large-scale usability testing.) Should the separation be branched out into other parts of the system, e.g., would it be useful to have a separate clipboard for every desktop? Finally, are there corresponding simple, intuitive models that would apply to other environments, such as the professionally administered server? Further research may help us to answer these questions.

References

[1] D. Elliot Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical report, MITRE Corporation, March 1976.

[2] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[3] Butler Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971.

[4] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[5] This is the Ctrl-Alt-Del feature, which provides a trusted path to an unforgeable screen.

[5] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[6] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karen A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[7] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, August 1999.

[8] Qun Zhong. Providing secure environments for untrusted network applications - with case studies using virtual vault and trusted sendmail proxy. In *Proceedings of Second IEEE International Workshop on Enterprise Security*, pages 277–283, Los Alamitos, CA, 1997. IEEE CS Press.

An Objectbase Schema Evolution approach to Windows NT Security

K. Barker
Advanced Database Systems and Applications Laboratory
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada

Raj Jayaplan, R. Peters
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

Abstract

The current security model for the Windows NT operating system is powerful and offers many valuable features. The *User Manager* provided by Windows NT is the primary method for the provision of security maintenance. Unfortunately, this tool does not offer several features that would make the end-user's task more intuitive. This paper demonstrates a new technique to support the security on a Windows NT platform. Our system supports at least the following features: (1) An object-oriented hierarchy, so roles and groups can be supported in a more automated way. (2) A more intuitive user interface so the administrative errors are less likely to be problematic. (3) Simplified security management on a Windows NT platform. (4) Avoids unnecessary creation of objects (users / group) and redundant granting / revoking of privileges. This paper discusses a new security model that has these features in addition to those currently available on Windows NT.

1. Introduction

One of the most challenging problems in managing large networked systems is the complexity of security administration. Some of the challenges of security administration are: depth of security, ease of access, sound management, protection of integrity, cost-effectiveness, secrecy and confidentiality of key software systems, databases, and data networks. Most of the security models currently used require a trade-off between the depth of security and ease of maintenance. The dilemma is that the more secure a system becomes, the more of a barrier that security becomes to the normal operations for which it is intended. Thus, a security model should be designed in such a way that it is transparent to the users. Further it must be easy to maintain and manage even if a very complex security model is required to ensure its proper functions.

The primary goal of our model is to provide a flexible set of operations that will make security management easier and more understandable. This research uses an Objectbase Management System (OBMS) because of its ability to handle the complex information with complex relationships often found in non-trivial security models. These models are often characterized by their dynamics. In other words, once a security model has been deployed, changing system and application requirements often demand that the model adapt. Fortunately, substantial research has been undertaken in recent years describing how to manage the dynamics exhibited by object-based systems. Ideally these changes to the model should occur while the system continues normal operation, as it is often undesirable or even impossible to stop the system to deploy new security policies. Our model proposes the use of *dynamic schema evolution*, which plays a vital role in object base management systems because of its ability to make changes to the database schema while applications are running. Typical changes that may be required are to the domain structure, the functionality of a particular application or to meet new performance requirements. This paper describes a security management model based on well-known schema evolution techniques from OBMSs [1].

The paper also contributes by demonstrating how these theoretical techniques can be applied to Windows NT. Our system implements the object oriented schema evolution strategy on the NT operating system as a way of demonstrating its correctness and utility.

2. Fundamentals

Three fundamental aspects need to be considered before we can turn our attention to the specifics of our research. The first of these is the work on schema evolution in object-oriented systems with particular focus on an *axiomatic model*. Secondly, work directly related to non-discretionary access rules that are often captured in *roles*. Finally, it is useful to consider other work attempting to provide a "new" interface to an existing systems security model. Each of these is briefly discussed below.

A few groups have undertaken schema evolution research but in the interest of space we will focus only on the one that is directly related to this paper. Peters and Özsu [5] describe a sound and complete axiomatic model for dynamic schema evolution in object-based systems that support the key features of types and inheritance. The model can infer all schema relationships from two sets associated with each type. These sets are the known as the *essential supertypes* and the *essential properties*. Formal definitions for these sets is beyond the scope of this paper but we will provide an intuitive definition. Essential supertypes are those supertypes in the class hierarchy that must be included in the definition of a type, while the essential properties are those properties that cannot be dropped as schema changes are made. We will return to these concepts in more detail later but the interested reader can find detailed information in Peters and Özsu [1]. This work also describes various dynamic schema policies used by TIGUKAT to support evolution and how these policies can be defined using axioms.

The second key foundation for this research is that of *role based access control* [6,7,8]. Ferrialo and Kuhn [3] describe a non-discretionary access control mechanism known as role based access control suitable for the needs of non-military systems. Ferrialo and Kuhn [4] argue that access control decisions are often based on the roles individual users adopt in the organization. Therefore, a role specifies a set of transactions that a user (or set of users) can perform within the context of that role in an organization.

Finally, we turn our attention to the problem of retrofitting a security interface on an existing system in the way similar to that proposed in this paper. Hua and Osborn [9] provide an interface between the role based access control and UNIX. A model of how to access UNIX files using the role based access control is also described. A *role graph* is used to visualize the permissions granted to the files in the UNIX system. However, to completely model the existing permissions in a UNIX environment, the system file permission and the links between the files must still be modeled for them to complete their research.

3. The Axiomatic Model

This section briefly reviews the relevant details of the axiomatic model used in this paper. We define schema evolution as the timely change of the schema and the consistent management of these changes. *Dynamic schema evolution* (DSE) is the management of schema changes while a system is in operation. The axiomatic model has been demonstrated to provide a method to support dynamic schema evolution in the objectbased system by serving as a common, formal underlying foundation for describing evolution in existing systems [1]. This suggests that we should be able to apply it to other systems that exhibit similar characteristics but before demonstrating that this is correct we must first define some key terms in the axiomatic model.

Type τ: Type in the axiomatic model defines the properties of objects. Types are used as templates for creating objects. An element of type τ is denoted as t

Type Lattice: The type lattice can be represented with a directed acyclic graph where the types are the graph's vertices and sub-type relationships are captured as directed edges.

Immediate supertype P(t): The immediate supertype of type t are those types that cannot be reached from t, transitively, through some other type.

Essential supertype P_e (t): Essential supertypes of a type t are those types that are essential in the construction and existence of type t.

Supertype Lattice L_t : Supertype lattice of type t is a set that includes t together with all the supertypes (immediate, essential or otherwise).

Native Properties N(t): of type t are those that are not defined in any of the t's supertypes.

Inherited Properties H(t): of type *t* is the union of the properties of all its supertypes.

Essential Properties $N_e(t)$: are those properties identified as being essential to the construction and existence of type *t*.

Interface I(t): of a type *t* is the union of native and inherited properties of type *t*.

Now we consider the basic operations common to schema evolution and security maintenance. Details and examples of each of these operations are available elsewhere [1] so we only provide the formal specifications in this paper.

Add a type: this operation adds a new type and integrates it with the existing lattice. The result of creating a new type *t* as the subtype of types $s_1, s_2 \ldots s_n$ with properties $P_1 \ldots P_m$ adds $s_1, s_2 \ldots s_n$ to $P_e(t)$, $P_1 \ldots P_m$ to $N_e(t)$ and the sets P(t), H(t), N(t), and I(t) are derived. If no supertypes are specified then T_object[1] is assumed.

Drop a Type: Removes a type from the schema. When a type is dropped it is removed from the P_e of all the subtypes of *t*.

Add Subtype Relationship: This operation adds a type as an essential supertype of another type, which effectively adds a subtype relationship between the two types. To add *s* as a supertype of *t*, *s* is added to $P_e(t)$ and the sets P(t), H(t), N(t) and I(t) are derived.

Drop a subtype Relationship: Removes a type as an essential supertype of another type, which effectively drops a subtype relationship between the two types. To drop type s as a supertype of *t*, *s* is removed from $P_e(t)$ and the sets P(t), H(t), N(t) and I(t) are derived. T_object cannot be removed as it is always essential.

Add a Property: Adds a property as an essential component of a type. To add a property *P* to type *t*, *P* is added to $N_e(t)$ and the sets N(t), H(t) and I(t) are derived.

Drop a Property: This operation drops a property as an essential component of a type. To drop a property *P* from type *t*, *P* is removed from $N_e(t)$ and the sets N(t), H(t) and I(t) are derived. Note that *P* is not removed from the interface of t because *P* may be inherited from one or more supertypes of *t*. However, if eventually the links to all supertypes that have *P* are removed, then *P* is no longer be part of *t*.

[1] T_object is the root class in the object hierarchy of the TIGUKAT model upon which we base this work.

4. Windows NT Security Model

Windows NT's security model is flat and does not support any hierarchical structure, let alone an object-oriented one. NT supports their security model with the *User Manager* [10]. The NT models security features supported by the *User Manager* include:

- Add / Remove a Group
- Add / Remove a User
- Add /Remove a member of a group
- Add / Remove privileges of a group
- Add / Remove privileges of a User

4.1 Add / Remove a group

NT's *User Manager* is used to add new groups to the system. Newly created groups do not have any privileges or user rights. A list of members can then be added to the group who then inherit the corresponding rights and privileges. This means that each user must be added to each group in which it should have privileges. It would be preferable to add groups of users to the newly created group thereby easing the process of creating new classes of users. In effect a hierarchy of user groups would be extremely helpful in security management.

Conversely, when a group is removed all members lose their membership. If the group has privileges, its members will all lose them unless they are explicitly given to the member through the granting of direct privilege. Ideally privileges could be grouped and formed into a hierarchy so that by dropping a subgroup the users would lose only a subset of privileges while maintaining those granted by the "super" group.

4.2 Add / Remove a User

New users initially belong to the "Users" group but the *User Manager* can insert them into additional groups. Users are atomic in that NT does not support the concept of a "user hierarchy" so users are inserted into groups only. When users are removed, they are physically removed from the system. The removed user is extracted from any groups to which they belonged. Finally, it should be noted that when a user is deleted it is completely removed from the system so even adding an identically named "new" user does not restore the old one.

4.3 Add / Remove Member to the group

Group membership can be specified while creating the group or user[2], or at any time after the group is created. Once a user is added to a group, all group privileges are inherited. Groups are composed of an arbitrary number of users but cannot contain other groups. In short, a group is only composed of existing users.

When a member is removed from a group it loses all privileges it inherited from the group except those that were granted directly. For example, if a user is given a privilege 'P' explicitly (direct privilege) which is also inherited from ones of its groups, the removal of the group does not remove 'P' because of the direct privilege.

4.4 Add / Remove privilege from a User

Users can have privileges granted to them from the *User Manager*. These are known as *direct privileges*. If this privilege is revoked the user will lose the direct privilege only..In other words, if the user is a member of a group that holds the privilege, the user will not lose it completely. This case is very difficult to handle with NT's *User Manager* as we discuss in the next section.

4.5 Add / Remove Privilege of a group

Whenever a privilege is added to a group it is propagated automatically to all members. Unfortunately Windows NT does not provide a mechanism to view privileges inherited as a result of group membership. The *User Manager* only provides a list of direct user privileges. Therefore, changes to group privileges are not reflected when viewing the users in the *User Manager*. A list of users/groups with a particular privilege can be created but it is impossible to find a complete list of privileges held by a particular user.

Consider a scenario where we would like to remove one of a user's privileges. To accomplish this we would like to remove the direct privilege and the privilege from all groups to which the user belongs that have the privilege. Deleting the direct privilege is trivial and once completed the *User Manager* will correctly display the absence of the privilege. But what about the privileges inherited from the groups? NT is very robust in that deleting a privilege from a group will remove it from both the group and its members. Although this is correct, it is impossible to tell by simply looking at the user privileges if the privilege has been completely removed. If we identify all groups but one containing the privilege, it will remain and be undetectable.

[2] Users can only be inserted at the time they are created if the group has already been created.

5. Modeling the Windows NT security with the Axiomatic model

The difficulties described above can be handled by treating users and groups as objects in an object-oriented hierarchy. We can then use the axiomatic model introduced earlier to manage the changes to these privileges. By treating these in a formal way we are able to ensure that the correct propagation of privileges occurs even if they are inherited in unexpected ways. Before discussing our implementation of this strategy we must first describe how Windows NT's security can be described with the axiomatic model.

5.1 Architecture Overview

Our security management system has three components, namely: the Windows NT layer, the axiomatic layer and the Security Manager Interface (see Figure 1).

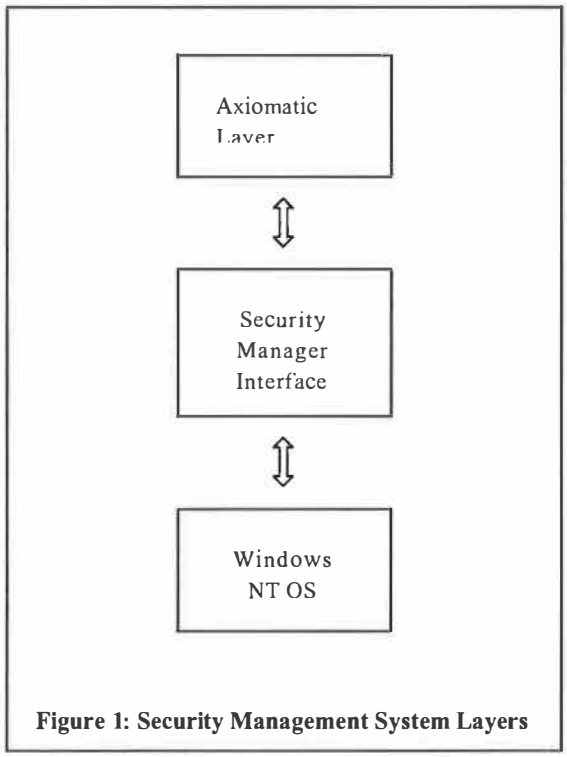


Figure 1: Security Management System Layers

The Security Manager Interface connects Windows NT's security system with the axiomatic model. Each component is discussed in the next few sections.

5.2 The Axiomatic Layer

Windows NT's security model is a flat structure. Users cannot be subtypes of others nor can a group be a subtype of another group. However, the same flat structure can be expressed with an axiomatic model thereby enabling both groups and users to be classified as types. The axiomatic model defines every type (users or groups) using subtypes and supertypes. Our system implements this object-oriented model with a file (external to NT's security system) that holds metadata about each of the types (users and groups). Therefore, the axiomatic layer is composed of two important components:

- The Axiomatic model
- The Metadata Axiom File (MAF)

NT user and group properties and their relationships are captured and expressed in terms of axiomatic properties. Any change to the subtype/supertype relationships or their properties result in corresponding change to the axiomatic model's properties. This means that all features supported by the axiomatic model such as: adding and dropping types, subtypes, supertypes and properties of a type are automatically supported by our security management system. Recall that Windows NT does not support these features because its security mechanism is not object-oriented.

The Metadata Axiom File is required because state information required by the axiomatic component is completely different from the structure of the traditional Windows NT security model. Thus the Metadata Axiom File holds all data required to build the axiomatic model of the NT defined users and groups.

Metadata stored in the MAF includes:

- All the types
- The subtypes of each type
- The supertype of each type
- Essential properties of all types
- Native properties contained in each type
- Inherited properties of all types

This metadata provides the information required to efficiently manage security on a NT machine. The Security Manager stores these properties in the MAF so all information about user and group state is saved. The axiomatic model for the user is dynamic so each time our Security Manager is loaded, the axiomatic model is built from the metadata in the MAF, thereby restoring the exact state of each of the objects.

Although Windows NT does not support the features provided by the axiomatic model, the Security Manager provides the various object-oriented features that enhances the security model.

5.3 The Security Manger Interface

The Security Manager Interface (Figure 1) ensures that all changes made by our tool are propagated to NT. It provides a translation from/to Windows NT's security model and the axiomatic model. Since our model sees security privilege changes as schema evolution, updates to NT privileges are propagated as axioms to the axiomatic model. Conversely changes within the Security Manager are propagated to NT as privilege changes.

5.4 System Operation

Our security model is installed above the native security model of Windows NT. The axiomatic model is represented by a directed graph where a node represents each user or group and the subtype/supertype relationship is an edge. An edge from node *A* to node *B* indicates that the type *A* (user/group) is the supertype of type *B*. Type *B* inherits all properties of *A* as expected.

For any two nodes N_i, N_j, if N_i is contained in the interface of N_j, then there must be a path between N_i and N_j. The interface of a particular node *A* is a set that contains all the nodes that are supertypes of *A*.

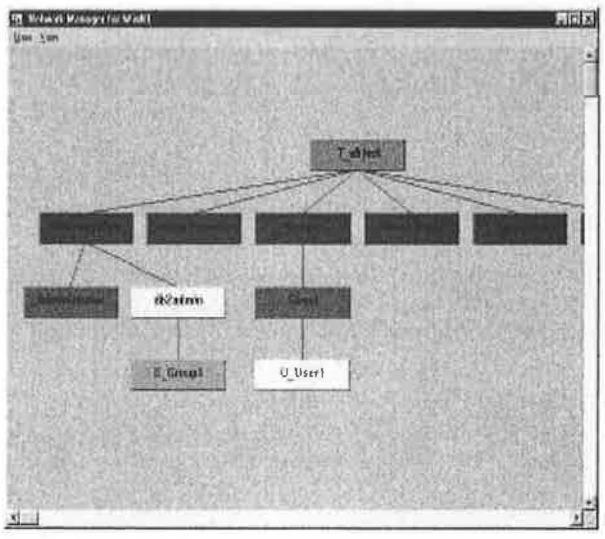


Figure 2: The Interface composed of groups and users

Figure 2 provides a snapshot of the object–oriented view of our security model for Windows NT. Type T_object is the root of all other types (both users and groups). This means type T_object has the minimum privilege and that all others inherit the privileges it possesses. The groups *Administrators, Power Users, Guests, Replicator* and *Users* are the immediate subtypes of T_Object. Since NT does not allow a group to be a member of another group, all groups are attached to T_object when the native security model is converted to the axiomatic model. The users *db2admin* and *Guest* are the members of the *Administrators* and *Guests* groups, respectively, so there is an edge from the type *Administrators* to the user *db2admin* and from *Guests* and *Guest*. For example, user *db2admin* and groups *G_Group1* and *Administrator* inherit all the properties of *Administrators*. Similarly the users *Guest* and *U_User1* inherit all the properties of the *Guests* group. To achieve greater clarity, different colors are given to System defined groups (Red), System defined users (Magenta), User defined groups (Green) and User defined Users (Yellow)[3].

We now summarize some of the axiomatic components defined for NT objects:

Type: NT objects are commonly referred to as types. This includes both users and groups.

Properties: The privileges associated with each group or user are termed *properties*.

Subtype and Supertype: Subtyping permits an object to be built based on another. For example, when a user is added as a member of a group, then we say that the user is a subtype of group or the group is the supertype of the user. Both users and groups are viewed as types so a user can be a subtype of another user (this prevents the unnecessary creation of groups in some cases), group can be a subtype of another group or a group can be a subtype of another user. By subtyping, an object (user or groups) inherits all the properties of the supertype. An object can have multiple supertypes and in that case it inherits the properties of all the supertypes[4].

Essential Supertype: The essential supertype of an object (user or group) contains all the users or groups that are essential to construct the object. All immediate supertypes are essential so every group is an essential supertype to its members.

Supertype Lattice: An object's type lattice contains the object and all its super-types.

We now turn our attention to the security management features supported by our security manager:

- Add / Remove a Group
- Add / Remove a User
- Add /Remove a sub-type (both users and groups)
- Add / Remove privileges of a group
- Add / Remove privileges of a User

Each of these is discussed in detail in the following sections.

5.4.1 Add / Remove Group

When a group is added, axiomatic metadata should be specified. This data might include the essential supertypes, immediate subtypes and the privileges the group possesses. Unlike NT's security model, ours allows the users or groups to be essential supertypes or immediate subtypes. In other words, both the users and groups can contain others. The group and its privileges are sent to the axiomatic component, which adds this group as a new type. Once the axiomatic model is updated, the affected NT objects (users/groups/privileges) are modified.

The insertion process requires that the group is created and all supertype privileges are given to this group. Thus, the group has inherited all the properties of its supertype. In cases where the properties of the supertypes overlap, only one copy of the properties are inherited thereby avoiding conflicts. These privileges must now be propagated to all its immediate subtypes. Two cases must be considered:

Case 1: If the immediate subtype is a user then the user is added to the group's membership roster. Once added to the group, NT propagates the group's privileges so there is no need to do this explicitly.

Case 2: If the immediate subtype is a group, then all of the group's privileges must be explicitly propagated to

[3] These appear as different shades of gray in this paper but we appeal to the readers imagination and intuition for the purpose of this submission. Demos of the system can be acquired by contacting the authors.

[4] This ability to subtype from both groups and users is extremely flexible. This expressive power is extremely useful but not all combinations of group/user subtyping will be required for NT. In fact, some may be irrelevant.

the subtype groups. This explicit propagation must be handled recursively to ensure that all children, grandchildren, etc. receive the necessary privileges.

Figure 3 (a) depicts groups *A, B, C, D, E* and *F* with a graphical depiction of the axiomatic model. Addition of group *X* as a supertype of *A* results in the propagation of *X*'s privileges (*P1, P2, P3, P4*) to the immediate subtype *A* and to groups *C, D, E* and *F* which is the behavior expected in an object-oriented inheritance hierarchy (see Figure 3 (b)).

Deleting a group removes the corresponding type from the axiomatic model. These changes are reflected to all its subtypes but the effect differs depending on the subtype's type (i.e. user or group). If the subtype is a user, it is removed from membership in this group. If it a group, the privilege is removed unless it is held as a direct privilege or inherited through another path. These changes are subsequently propagated recursively to all the affected types (groups and users) below this point in the hierarchy.

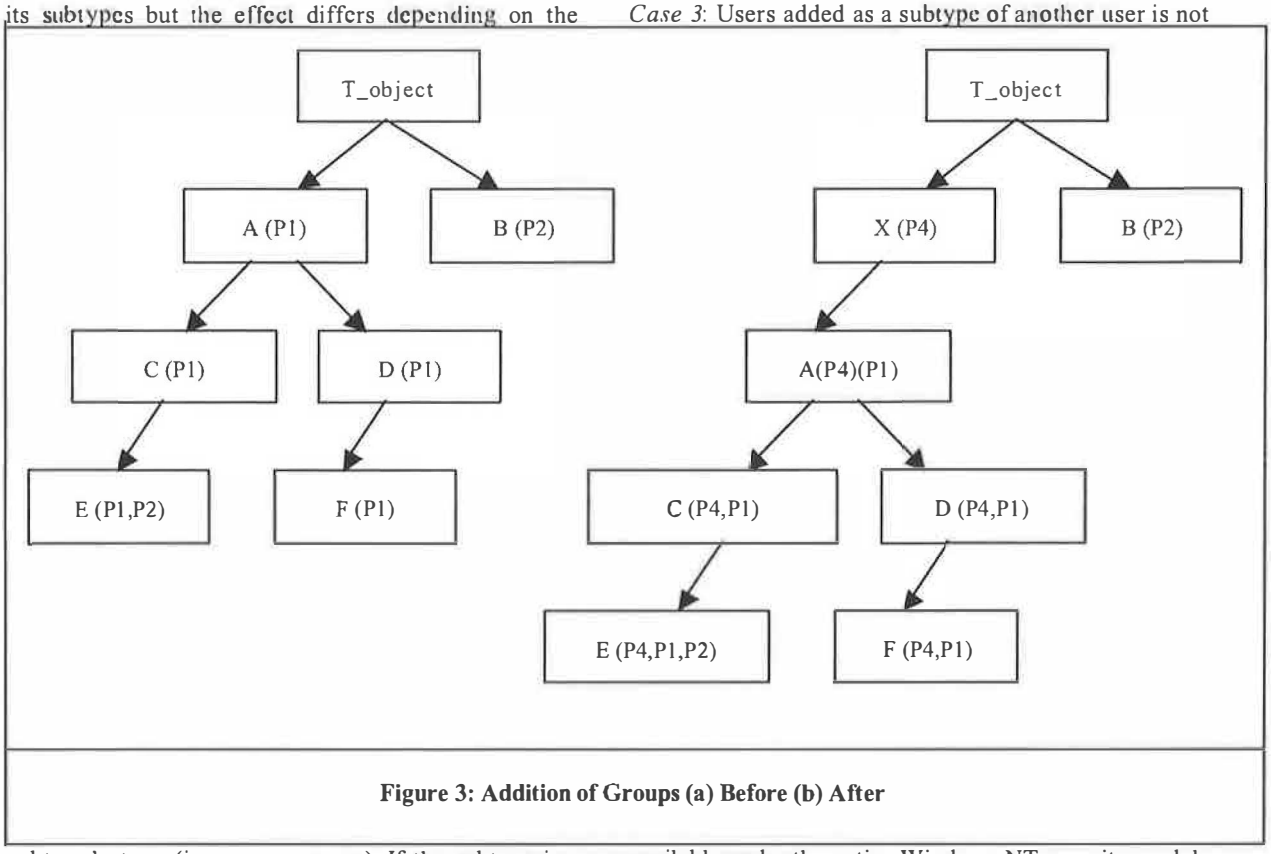


Figure 3: Addition of Groups (a) Before (b) After

5.4.2 Add / Remove a User

Adding a user to our system results in the creation of the new user on NT. The user can be created with no supertypes (except *T_object*) or as a subtype of a group or user. Once again several cases need to be considered:

Case 1: Users without supertypes are subtypes of *T_object*. This is consistent with the axiomatic model because every type must be a subtype of *T_Object*.

Case 2: Users added as a subtype of a group (or many groups) are made members of each supertype group. In this way the user inherits all the privileges of the supertype, which is consistent with the object-oriented model.

Case 3: Users added as a subtype of another user is not available under the native Windows NT security model. This feature would prevent the unnecessary creation of groups in cases, where a user needs to possess all the properties of a different user. This operation would permit a user to inherit all privileges of a particular user. Additional research would need to be undertaken to define the semantics of removing a supertype of a user, but the issue is beyond the scope of this paper.

Our Privilege Propagation algorithm carries out this propagation for all cases.

If a user is deleted from the system, the axiomatic model is first updated and the corresponding effects are propagated to the NT objects. The Privilege Propagation algorithm (modified to revoke privilege) removes privilege from the object's subtypes. Finally, the user looses membership in the supertype's groups.

The algorithm used to add and remove groups and users is described in the Algorithm 1.

Algorithm 1: Addition or Removal of a group or user

1. Update the axiomatic components including
    Essential Super-type
    Immediate sub-type
    Type Lattice
2. If X is a User **Then**
    Delete / Create X as User
        If the operation is addition **Then**
        Add X as the member of all its super-type
        Endif
  Else
    Delete / Create X as a new group
    Revoke / Grant all the privileges of its super-types to X
  Endif
3. Initialize the groupList with all the sub-types of X
4. For each element in the groupList **do**
    If the element I is a user **Then**
        Remove / Add this user I as a member tothe group X
    Else
        revoke / grant the privilege of group X to this group I
    Endif
    For each sub-type of the group I **do**
        Add this sub-type to the groupList
    Endfor
    Remove the element I from the list
    If the groupList is empty **Then**
      Return
    Endif
  Endfor
End Algorithm 1

5.4.3 Modify Subtype Relationship

Addition (removal) of an object (user or group) as a supertype of another object results in the addition (removal) of the set of essential supertype for the object too. Our model permits the supertype to be either a user or group. As in previous cases, once the axiomatic model is updated the process of privilege propagation and addition/deletion of users as members is performed on the NT machine itself.

5.4.4 Add/Remove Privilege of a group

We now turn out attention to the specific issue of privilege management used in our system. Before presenting the details of our implementation we provide a few definitions[5]. Privileges in our model are broadly classified into two three types:

- *Native Privilege*: are directly given to the types
- *Inherited Privilege*: are acquired by the types from the parent (user or group)
- *Privilege Interface*: the union of the inherited and the native privileges

Our system clearly presents a list of each privilege held by an object as illustrated in Figure 4.

Figure 4 depicts user *U_User1* membership in Guests and that he holds the 'Shutdown the System' privilege. The inherited privilege 'Log on to local machine' would not appear as an NT privilege (privileges as seen using the *User Manager*) because these are not displayed.

The addition and removal of privileges is the final aspect of our system to consider. The key issues here are related to how privileges are propagated throughout the object hierarchy and how these are passed onto the flat structure found in NT. A brief discussion of the addition and removal of privileges is provided followed immediately by a sketch of the algorithm that implements this aspect of our system.

Add a Privilege: When a privilege is added to a type, it is added to the set of native privileges. The privilege must be propagated to its sub-types. The process of privilege propagation is similar for both users and groups. These privileges only need to be granted to the groups because NT propagates them to the group's members on our systems behalf.

Remove a Privilege: Removing a privilege from a user or group requires it first be removed from the set of direct privileges. In our system, if the privilege is found in the set of inherited privileges it has been acquired from at

[5] We have used these terms intuitively early in the paper but we need a more precise definition to describe this aspect of our implmentation.

least one of its parents. This means the privilege is not revoked from the Windows NT system and to do so would require that the corresponding privilege be removed from the parent. Even in the native NT model it is not possible to remove the inherited privilege. Only direct privileges can be deleted and this property is not changed in our model too. Alternatively, the object could be removed from the parent carrying the privilege (see Section 5.4.3). Once removed the privilege must be recursively applied to its subtypes.

The system has been implemented and proven to provide an excellent intuitive interface that meets the primary goals of our project. The system is sufficiently flexible that a system administrator can use our system to deploy new roles, groups and users but if they choose to use the *User Manager* to complete a task, our system will resynchronize with those changes once it is restarted. In this way, both our system and the one provided by NT can be used for complementary tasks. We believe however that our model is much more intuitive and it should be further investigated with the ultimately goal of deploying it as native to NT.

6. Conclusion

This paper describes a new security management model based on well-known schema evolution techniques in OBMSs. The model is successfully implemented on Windows NT above the original security model in such a way that it does not require modification or introduce conflicts to NT's current approach. We believe that one of the nicest features of our approach is that both our system and the *User Manager* can operate together. Any changes made with the *User Manager* are reflected in our system and changes done using our tool can be seen with the *User Manager* in precisely the way you would expect.

Several drawbacks associated with Windows NT's security model and its maintenance tool (the User Manager) are addressed by this research including:

- Failure to provide a clear link between inherited privileges arising from participation in groups and the lack of a technique to extract this information in an easy clear way.
- Inherited privileges of the members are hidden.
- Lack of easier mechanism to find a complete list of privileges held by a particular user/group.
- Lack of easier mechanism to find out the various groups to which each user belongs.
- An improved visual interface to the security model.

We have also demonstrated how our model permits the user to perform various operations that makes the management of security easier and more understandable. Some of the benefits of the new model include:

- Avoids unnecessary creation of groups.
- Prevents redundant and unnecessary granting and revoking of privileges.
- Provides a better visual interface that clearly illustrates the privilege flow.

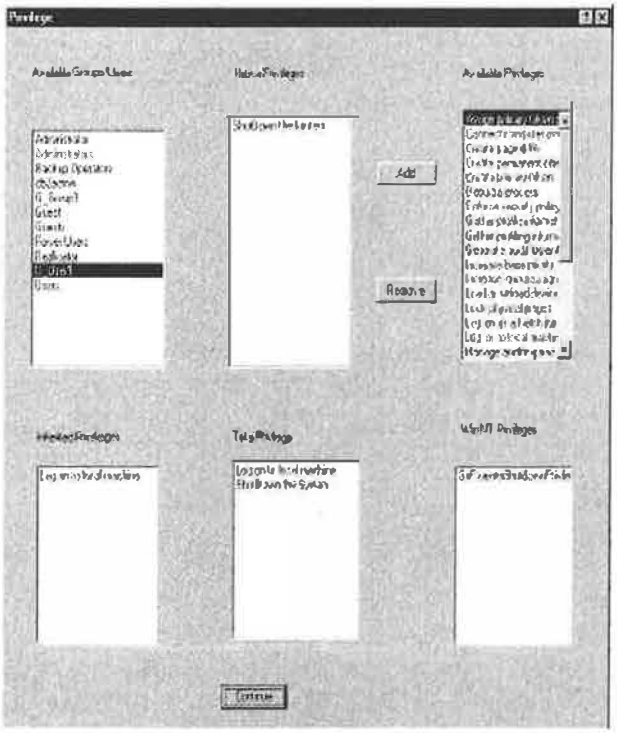


Figure 4: Interface Showing the Available Privileges

- Features of the object-oriented model that enhances the maintenance of the security model are used.
- Grants no more privilege than is necessary to perform a task. This property ensures the adherence to the security principle of least privilege.

References

[1] R. J. Peters and M. T. Özsu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Management Systems", ACM Transactions on Database Systems, 22(1): 75-114, March 1997.

Algorithm 2: Addition / removal of privileges

```
1. Add / Remove the privilege P from the set Native Privilege of type T
2. If privilege P not found in set Inherited Privilege of type T Then
      Add / Remove the privilege P from the type T (can be user or group)
      If T a group Then
         Add all the sub groups of the type T to the GroupList
      Else
         Add all the sub-types (both users and groups) of T to the GroupList
      Endif
      For each element I in the GroupList do
          Add / Remove the privilege P from the set Inherited Privilege
          If privilege P not found in set Direct Privilege Then
             Add / Remove the privilege P from the type T (can be user or group)
             If T a group Then
                Add all the sub groups of the type T to the GroupList
             Else
                Add all the sub-types (both users and groups) of T to the GroupList
             Endif
          Endif
          Remove the element I from the list
          If the groupList is empty Then return
      Endfor
  Endif
EndAlgorithm 2
```


[2] R. J. Peters. TIGUKAT: A Uniform Behavioral Objectbase Management System. Ph.D thesis. University of Alberta. TR94 – 06. April 1994

[3] D. Ferrialo and R. Kuhn. Role Based Access Control. 15th National Computer Security Conference. 1992.

[4] D. Ferrialo, A. Cugini, and R. Kuhn. Role Based Access Control : Features and Motivation . Computer Security Application Conference. 1995.

[5] R.J.Peters and M.T.Özsu. Axiomatization of Dynamic Schema Evolution in Objectbases, In 11th International Conference on Data Engineering, Taiwan, March 1995.

[6] J.Barkley. Implementing Role Based Access Control Using Object Technology. First ACM Workshop on Role Based Access Control. November 1995.

[7] J.Barkley. Comparing Simple role Based Access Control Models and Access Control Lists. Second ACM Workshop on Role Based Access Control. August 1997.

[8] J.Barkley and A.Cincotta. Managing Role/Permission Relationships Using Object Access Types. Third ACM Workshop on Role Based Access Control. July 1998.

[9] L.Hua and S.Osborn. Modeling UNIX access control with a Role Graph. In the Proceedings of the Ninth International Conference on Computing and Information. Pages 131 –138. June 1998.

[10] M.Minasi, C.Anderson, E.Creegan. Mastering Windows NT Server. Fourth Edition. 1997.

An Empirical Study of the Robustness of Windows NT Applications Using Random Testing

Justin E. Forrester Barton P. Miller
{jforrest,bart}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

We report on the third in a series of studies on the reliability of application programs in the face of random input. In 1990 and 1995, we studied the reliability of UNIX application programs, both command line and X-Window based (GUI). In this study, we apply our testing techniques to applications running on the Windows NT operating system. Our testing is simple black-box random input testing; by any measure, it is a crude technique, but it seems to be effective at locating bugs in real programs.

We tested over 30 GUI-based applications by subjecting them to two kinds of random input: (1) streams of valid keyboard and mouse events and (2) streams of random Win32 messages. We have built a tool that helps automate the testing of Windows NT applications. With a few simple parameters, any application can be tested.

Using our random testing techniques, our previous UNIX-based studies showed that we could crash a wide variety of command-line and X-window based applications on several UNIX platforms. The test results are similar for NT-based applications. When subjected to random valid input that could be produced by using the mouse and keyboard, we crashed 21% of applications that we tested and hung an additional 24% of applications. When subjected to raw random Win32 messages, we crashed or hung all the applications that we tested. We report which applications failed under which tests, and provide some analysis of the failures.

1 Introduction

We report on the third in a series of studies on the reliability of application programs in the face of random input. In 1990 and 1995, we studied the reliability of UNIX command line and X-Window based (GUI) application programs[8,9]. In this study, we apply our techniques to applications running on the Windows NT operating system. Our testing, called *fuzz* testing, uses simple black-box random input; no knowledge of the application is used in generating the random input.

Our 1990 study evaluated the reliability of standard UNIX command line utilities. It showed that 25-33% of such applications crashed or hung when reading random input. The 1995 study evaluated a larger collection of applications than the first study, including some common X-Window applications. This newer study found failure rates similar to the original study. Specifically, up to 40% of standard command line UNIX utilities crashed or hung when given random input and 25% of the X-Window applications tested failed to deal with the random input. In our current (2000) study, we find similar results for applications running on Windows NT.

Our measure of reliability is a primitive and simple one. A program passes the test if it responds to the input and is able to exit normally; it fails if it crashes (terminated abnormally) or hangs (stops responding to input within a reasonable length of time). The application does not have to respond sensibly or according to any formal specification. While the criterion is crude, it offers a mechanism that is easy to apply to any application, and any cause of a crash or hang should not be ignored in any program. Simple fuzz testing does not replace more extensive formal testing procedures. But curiously, our simple testing technique seems to find bugs that are not found by other techniques.

Our 1995 study of X-Window applications provided the direction for the current study. To test X-Window applications, we interposed our testing program between the application (client) and the X-window display server. This allowed us to have full control of the input to any application program. We were able to send completely random messages to the application and also to send random streams of valid keyboard and mouse events. In our current Windows NT study, we are able to accomplish the same level of input control of an application by using the Windows NT event mechanisms (described in Section 2).

Subjecting an application to streams of random valid keyboard and mouse events tests the application under conditions that it should definitely tolerate, as they could occur in normal use of the software. Subjecting an application to completely random (often invalid) input messages is a test of the general strength of error checking. This might be considered an evaluation of the software engineering discipline, with respect to error handling, used in producing the application.

Five years have passed since our last study, during which time Windows-based applications have clearly come to dominate the desktop environment. Windows NT (and now Windows 2000) offers the full power of a modern operating system, including virtual memory, processes, file protection, and networking. We felt it was time to do a comparable study of the reliability of applications in this environment.

Our current study has produced several main results:

- 21% of the applications that we tested on NT 4.0 *crashed* when presented with random, valid keyboard and mouse events. Test results for applications run on NT 5.0 (Windows 2000) were similar.
- An additional 24% of the applications that we tested *hung* when presented with random valid keyboard and mouse events. Tests results for applications run on NT 5.0 (Windows 2000) were similar.
- Up to 100% of the applications that we tested failed (crashed or hung) when presented with completely random input streams consisting of random Win32 messages.
- We noted (as a result of our completely random input testing) that *any* application running on Windows platforms is vulnerable to random input streams generated by any other application running on the same system. This appears to be a flaw in the Win32 message interface.
- Our analysis of the two applications for which we have source code shows that there appears to be a common careless programming idiom: receiving a Win32 message and unsafely using a pointer or handle contained in the message.

The results of our study are significant for several reasons. First, reliability is the foundation of security[4]; our results offer an informal measure of the reliability of commonly used software. Second, we expose several bugs that could be examined with other more rigorous testing and debugging techniques, potentially enhancing software producers' ability to ship bug free software. Third, they expose the vulnerability of applications that use the Windows interfaces. Finally, our results form a quantitative starting point from which to judge the relative improvement in software robustness.

In the 1990 and 1995 studies, we had access to the source code of a large percentage of the programs that we tested, including applications running on several vendors' platforms and GNU and Linux applications. As a result, in addition to causing the programs to hang or crash, we were able to debug most applications to find the cause of the crash. These causes were then categorized and reported. These results were also passed to the software vendors/authors in the form of specific bug reports. In the Windows environment, we have only limited access (thus far) to the source code of the applications. As a result, we have been able to perform this analysis on only two applications: emacs, which has public source code, and the open source version of Netscape Communicator (Mozilla).

Section 2 describes the details of how we perform random testing on Windows NT systems. Section 3 discusses experimental method and Section 4 presents the results from those experiments. Section 5 offers some analysis of the results and presents associated commentary. Related work is discussed in Section 6.

2 Random Testing on the Windows NT Platform

Our goal in using random testing is to stress the application program. This testing required us to simulate user input in the Windows NT environment. We first describe the components of the kernel and application that are involved with processing user input. Next, we describe how application programs can be tested in this environment.

In the 1995 study of X-Window applications, random user input was delivered to applications by inserting random input in the regular communication stream between the X-Window server and the application. Two types of random input were used: (1) random data streams and (2) random streams of valid keyboard and mouse events. The testing using random data streams sent completely random data (not necessarily conforming to the window system protocol) to an application. While this kind of input is unlikely under normal operating conditions, it provided some insight into the level of testing and robustness of an application. It is crucial for a properly constructed program to check values obtained from system calls and library routines. The random valid keyboard and mouse event tests are essentially testing an application as though a monkey were at the keyboard and mouse. Any user could generate this input, and any failure in these circumstances represents a bug that can be encountered during normal use of the application.

We used the same basic principles and categories in the Windows NT environment, but the architecture is slightly different. Figure 1 provides a simplified view of the components used to support user input in the Windows NT environment[10,11,12].

We use an example to explain the role of each component in Figure 1. Consider the case where a user clicks on a link in a web browser. This action sets into motion the Windows NT user input architecture. The

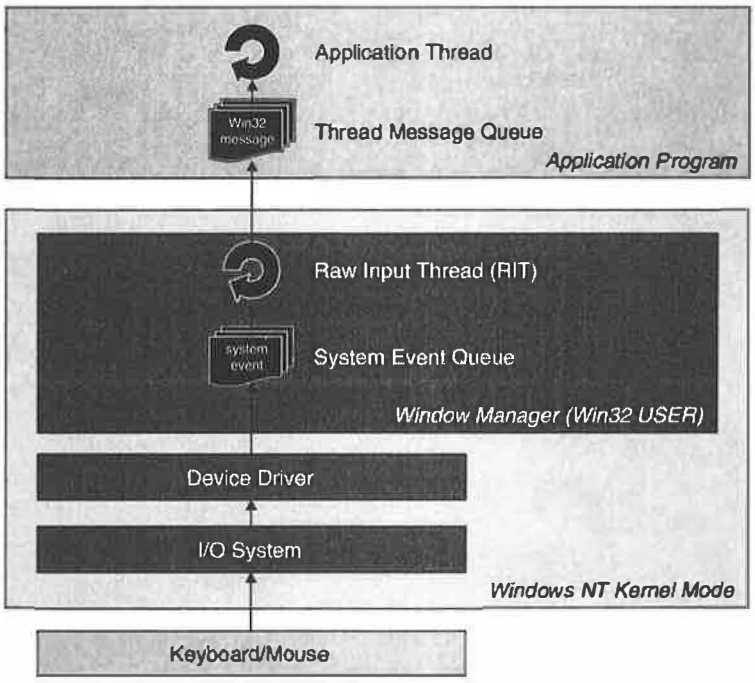


Figure 1: Windows NT Architectural Components for User Input

mouse click first generates a processor interrupt. The interrupt is handled by the I/O System in the base of the Windows NT kernel. The I/O System hands the mouse interrupt to the mouse device driver. The device driver then computes the parameters of the mouse click, such as which mouse button has been clicked, and adds an event to the System Event Queue (the event queue of the Window Manager) by calling the `mouse_event` function. At this point, the device driver's work is complete and the interrupt has been successfully handled.

After being placed in the System Event Queue, the mouse event awaits processing by the kernel's Raw Input Thread (RIT). The RIT first converts the raw system event to a Win32 message. A Win32 message is the generic message structure that is used to provide applications with input. The RIT next delivers the newly created Win32 message to the event queue associated with the window. In the case of the mouse click, the RIT will create a Win32 message with the `WM_LMOUSEBUTTONDOWN` identifier and current mouse coordinates, and then determine that the target window for the message is the web browser. Once the RIT has determined that the web browser window should receive this message, it will call the `PostMessage` function. This function will place the new Win32 message in the message queue belonging to the application thread that created the browser window.

At this point, the application can receive and process the message. The Win32 Application Program Interface (API) provides the `GetMessage` function for applications to retrieve messages that have been posted to their message queues. Application threads that create windows generally enter a "message loop". This loop usually retrieves a message, does preliminary processing, and dispatches the message to a registered callback function (sometimes called a *window procedure*) that is defined to process input for a specific window. In the case of the web browser example, the Win32 message concerning the mouse click would be retrieved by the application via a call to `GetMessage` and then dispatched to the window procedure for the web browser window. The window procedure would then examine the parameters of the `WM_LMOUSEBUTTONDOWN` message to determine that the user had clicked the left mouse button at a given set of coordinates in the window and that the click had occurred over the web link.

Given the above architecture, it is possible to test applications using both random events and random Win32 messages. Testing with random events entails inserting random system events into the system event queue. Random system events simulate actual keystroke or mouse events. They are added to the system via the same mechanism that the related device driver uses,

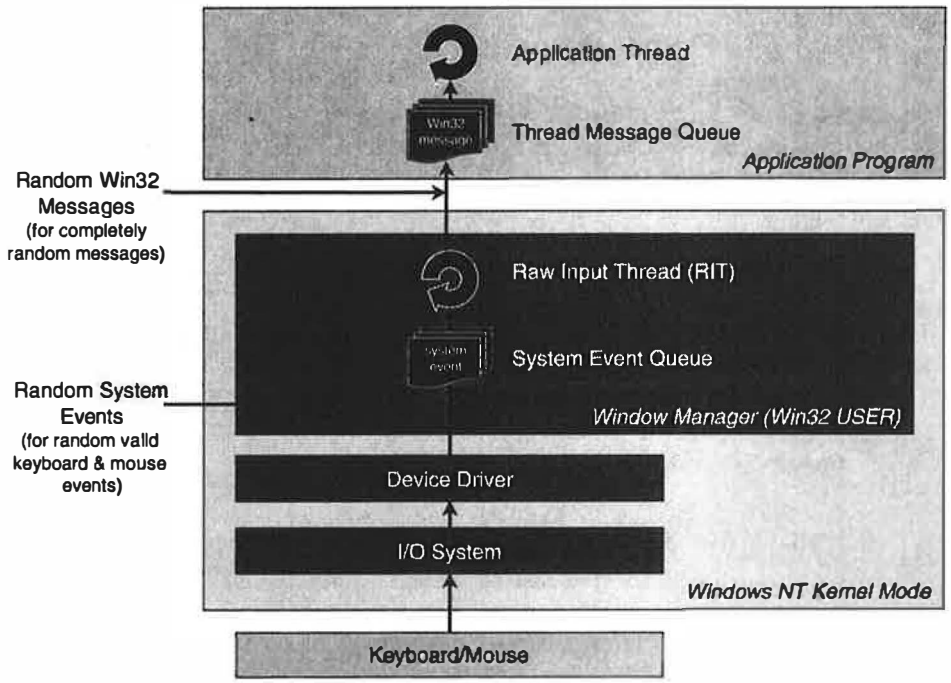


Figure 2: Insertion of Random Input

namely the `mouse_event` and `keybd_event` functions.

The second random testing mechanism involves sending random Win32 messages to an application. Random Win32 messages combine random but valid message types with completely random contents (parameters). Delivering these messages is possible by using the Win32 API function `PostMessage`. The `PostMessage` function delivers a Win32 message to a message queue corresponding to a selected window and returns. Note that there is similar function to `PostMessage`, called `SendMessage`, that delivers a Win32 message and waits for the message to be processed fully before returning. Win32 messages are of a fixed size and format. These messages have three fields, a message ID field and two integer parameters. Our testing produced random values in each of these fields, constraining the first field (message ID) to the range of valid message ID's.

Figure 2 shows where each random testing mechanism fits into the Windows NT user input architecture.

Notice in Figure 2 that under both testing conditions, the target application is unable to distinguish messages sent by our testing mechanisms from those actually sent by the system. This distinction is essential to create an authentic test environment.

3 Experimental Method

We describe the applications that we tested, the test environment, our new testing tool (called *fuzz*), and the tests that we performed. We then discuss how the data was collected and analyzed.

3.1 Applications and Platform

We selected a group of over 30 application programs. While we tried to select applications that were representative of a variety of computing tasks, the selection was also influenced by what software was commonly used in the Computer Sciences Department at the University of Wisconsin. The software includes word processors, Web browsers, presentation graphics editors, network utilities, spread sheets, software development environments, and others. In addition to functional variety, we also strove to test applications from a variety of vendors, including both commercial and free software.

The operating system on which we ran and tested the applications was Windows NT 4.0 (build 1381, service pack 5). To insure that our results were timely, we tested a subset of the applications on the new Windows 2000 system (version 5.00.2195). For the 14 applications that we re-tested on Windows 2000, we obtained similar results to those tested under NT 4.0. The hard-

ware platform used for testing was a collection of standard Intel Pentium II PCs.

3.2 The Fuzz Testing Tool

The mechanism we used for testing applications was a new tool, called *fuzz*, that we built for applications running on the Windows NT platform. Fuzz produces repeatable sequences of random input and delivers them as input to running applications via the mechanisms described in Section 2. Its basic operation is as follows:

1. Obtain the process ID of the application to be tested (either by launching the application itself or by an explicit command line parameter).
2. Determine the main window of the target application along with its desktop placement coordinates.
3. Using one of `SendMessage`, `PostMessage`, or `keybd_event` and `mouse_event`, deliver random input to the running application.

Fuzz is invoked from a command line; it does not use a GUI so that our interactions with the tool do no interfere with the testing of the applications. The first version of our Windows NT fuzz tool had a GUI interface but the use of the GUI for the testing tool interfered with the testing of the applications. As a result, we changed fuzz to operate from a command line. The fuzz command has the following format:

```
fuzz [-ws] [-wp] [-v] [-i pid] [-n
#msgs] [-c] [-l] [-e seed] [-a appl cmd
line]
```

Where `-ws` is random Win32 messages using `SendMessage`, `-wp` is random Win32 messages using `PostMessage`, and `-v` is random valid mouse and keyboard events. One of these three options must be specified.

The `-i` option is used to start testing an already-running application with the specified process ID, and `-a` tells fuzz to launch the application itself. The `-n` option controls the maximum number of messages that will be sent to the application, and `-e` allows the seed for the random number generator to be set.

The `-l` and `-c` options provide finer control of the `SendMessage` and `PostMessage` tests, but were not used in the tests that we report in this paper. Null parameters can be included in the tests with `-l` and `WM_COMMAND` messages (control activation messages such as button clicks) can be included with `-c`.

3.3 The Tests

Our tests were divided into three categories according to the different input techniques described in Section 2. As such, the application underwent a battery of random tests that included the following:

- 500,000 random Win32 messages sent via the `SendMessage` API call,
- 500,000 random Win32 messages sent via the `PostMessage` API call, and
- 25,000 random system events introduced via the `mouse_event` and `keybd_event` API calls.

The first two cases use completely random input and the third case uses streams of valid keyboard and mouse events.

The quantity of messages to send was determined during preliminary testing. During that testing, it appeared that if the application was going to fail at all, it would do so within the above number of messages or events. Each of the three tests detailed above was performed with two distinct sequences of random input (with different random seeds), and three test trials were conduced for each application and random sequence, for a total of 18 runs for each application. The same random input streams were used for each application.

4 Results

We first describe the basic success and failure results observed during our tests. We then provide analysis of the cause of failures for two applications for which we have source code.

4.1 Quantitative Results

The outcome of each test was classified in one of three categories: the application crashed completely, the application hung (stopped responding), or the application processed the input and we were able to close the application via normal application mechanisms. Since the categories are simple and few, we were able to categorize the success or failure of an application through simple inspection. In addition to the quantitative results, we report on diagnosis of the causes of the crashes for the two applications for which we have source code.

Figure 3 summarizes the results of the experiments for Windows NT 4.0 and Figure 4 has results for a subset of the applications tested on Windows 2000. If an application failed on any of the runs in a particular category (column), the result is listed in the table. If the application neither crashed nor hung, it passed the tests (and has no mark in the corresponding column).

The overall results of the tests show that a large number of applications failed to deal reasonably with random input. Overall, the failure rates for the Win32 message tests were much greater than those for the random valid keyboard and mouse event tests. This was to be expected, since several Win32 message types include pointers as parameters, which the applications appar-

Application	Vendor	SendMessage	PostMessage	Random Valid Events
Access 97	Microsoft	●	●	○
Access 2000	Microsoft	●	●	○
Acrobat Reader 4.0	Adobe Systems	●	●	
Calculator 4.0	Microsoft		●	
CD-Player 4.0	Microsoft	●	●	
Codewarrior Pro 3.3	Metrowerks	●	●	●
Command AntiVirus 4.54	Command Software Systems	●	●	
Eudora Pro 3.0.5	Qualcomm	●	●	○
Excel 97	Microsoft	●	●	
Excel 2000	Microsoft	●	●	
FrameMaker 5.5	Adobe Systems		●	
FreeCell 4.0	Microsoft	●	●	
Ghostscript 5.50	Aladdin Enterprises	●	●	
Ghostview 2.7	Ghostgum Software Pty	●	●	
GNU Emacs 20.3.1	Free Software Foundation	●	●	
Internet Explorer 4.0	Microsoft	●	●	●
Internet Explorer 5.0	Microsoft	●	●	
Java Workshop 2.0a	Sun Microsystems		●	○
Netscape Communicator 4.7	Netscape Communications	●	●	●
NotePad 4.0	Microsoft	●	●	
Paint 4.0	Microsoft	●	●	
Paint Shop Pro 5.03	Jasc Software		○	
PowerPoint 97	Microsoft	○	○	○
PowerPoint 2000	Microsoft	○		○
Secure CRT 2.4	Van Dyke Technologies	●	●	○
Solitaire 4.0	Microsoft		●	
Telnet 5 for Windows	MIT Kerberos Group		●	
Visual C++ 6.0	Microsoft	●	●	●
Winamp 2.5c	Nullsoft	○	●	
Word 97	Microsoft	●	●	●
Word 2000	Microsoft	●	●	●
WordPad 4.0	Microsoft	●	●	●
WS_FTP LE 4.50	Ipswitch	●	●	○
Percent Crashed		72.7%	90.9%	21.2%
Percent Hung		9.0%	6.0%	24.2%
Total Percent Failed		81.7%	96.9%	45.4%

Figure 3: Summary of Windows NT 4.0 Test Results

● = *Crash*, ○ = *Hang*.

Note that if an application both crashed and hung, only the crash is reported.

ently de-reference blindly. The NT 4.0 tests using the `SendMessage` API function produced a crash rate of over 72%, 9% of the applications hung, and a scant 18% successfully dealt with the random input. The tests using the `PostMessage` API function produced a slightly higher crash rate of 90% and a hang rate of 6%. Only one application was able to successfully withstand the `PostMessage` test.

The random valid keyboard and mouse event results, while somewhat improved over the random Win32 message test, produced a significant number of

Application	Vendor	SendMessage	PostMessage	Random Valid Events
Access 97	Microsoft	●	●	
Access 2000	Microsoft	●	●	●
Codewarrior Pro 3.3	Metrowerks			●
Excel 97	Microsoft	●	●	
Excel 2000	Microsoft	●	●	
Internet Explorer 5	Microsoft	●	●	
Netscape Communicator 4.7	Netscape Communications	●	●	●
Paint Shop Pro 5.03	Jasc Software			○
PowerPoint 97	Microsoft	○		○
PowerPoint 2000	Microsoft	○		○
Secure CRT 2.4	Van Dyke Technologies	●	●	
Visual C++ 6.0	Microsoft	●	●	●
Word 97	Microsoft	●	●	●
Word 2000	Microsoft	●	●	●
Percent Crashed		71.4%	71.4%	42.9%
Percent Hung		14.3%	0.0%	21.4%
Total Percent Failed		85.7%	71.4%	64.3%

Figure 4: Summary of Windows 2000 Test Results
● = *Crash*, ○ = *Hang*.
Note that if an application both crashed and hung, only the crash is reported.

crashes. Fully 21% of the applications crashed and 24% hung, leaving only 55% of applications that were able to successfully deal with the random events. This result is especially troublesome because these random events could be introduced by any user of a Windows NT system using only the mouse and keyboard.

The Windows 2000 tests have similar results to those performed on NT 4.0. We had not expected to see a significant difference between the two platforms, and these results confirm this expectation.

4.2 Causes of Crashes

While source code was not available to us for most applications, we did have access to the source code of two applications: the GNU Emacs text editor and the open source version of Netscape Communicator (Mozilla). We were able to examine both applications to determine the cause of the crashes that occurred during testing.

Emacs Crash Analysis

We examined the emacs application after it crashed from the random Win32 messages. The cause of the crash was simple: casting a parameter of the Win32 message to a pointer to a structure and then trying to de-reference the pointer to access a field of the structure. In the file `w32fns.c`, the message handler (`w32_wnd_proc`) is a standard Win32 callback function. This callback function tries to de-reference its third parameter (`lparam`); note that there is no error checking or exception handling to protect this de-reference.

```
LRESULT CALLBACK
w32_wnd_proc (hwnd, msg, wParam, lParam)
{
   . . .
   POINT *pos;
   pos = (POINT *)lParam;
   . . .
   if (TrackPopupMenu((HMENU)wParam,
      flags, pos->x, pos->y, 0, hwnd,
      NULL))
      . . .
}
```

The pointer was a random value produced by fuzz, and therefore was invalid; this de-reference caused an access violation. It is not uncommon to find failures caused by using an unsafe pointer; our previous studies found such cases, and these cases are also well-documented in the literature [13]. From our inspection of other crashes (based only on the machine code), it appears that this problem is the likely cause of many of the random Win32 message crashes.

Mozilla Crash Analysis

We also examined the open source version of Netscape Communicator, called Mozilla, after it crashed from the random Win32 messages. The cause of the crash was similar to that of the emacs crash. The crash occurred in file `nsWindow.cpp`, function `nsWindow::ProcessMessage`. This function is designed to respond to Win32 messages posted to the application's windows. In fashion similar to the GNU emacs example, a parameter of the function (`lParam` in this case) is assumed to be a valid window handle.

```
. . .
nsWindow* control =
   (nsWindow*)::GetWindowLong(
      (HWND)lParam, GWL_USERDATA);
if (control) {
   control->SetUpForPaint(
   (HDC)wParam);
. . .
```

The value is passed as an argument to the `GetWindowLong` function, which is used to access application specific information associated with a particular window. In this case, the parameter was a random value produced by fuzz, so the `GetWindowLong` function is retrieving a value associated with a random window. The application then casts the return value to a pointer and attempts to de-reference it, thereby causing the application to crash.

5 Analysis and Conclusions

The goal of this study was to provide a first look at the general reliability of a variety of application programs running on Windows NT. We hope that this study inspires the production of more robust code. We first discuss the results from the previous section then provide some editorial discussion.

The tests of random valid keyboard and mouse events provide the best sense of the relative reliability of application programs. These tests simulated only random keystrokes, mouse movements, and mouse button clicks. Since these events could be caused by a user, they are of immediate concern. The results of these tests show that many commonly-used desktop applications are not as reliable as one might hope.

The tests that produced the greatest failure rates are the random Win32 message tests. In the normal course of events, these messages are produced by the kernel and sent to an application program. It is unlikely (though not impossible) that the kernel would send messages with invalid values. Still, these tests are interesting for two reasons. First, they demonstrate the vulnerability of this interface. Any application program can send messages to any other application program. There is nothing in the Win32 interface that provides any type of protection. Modern operation systems should provide more durable firewalls. Second, these results point to a need for more discipline in software design. Major interfaces between application software components and between the application and the operating system should contain thorough checks of return values and result parameters. Our inspection of crashes and the diagnosis of the source code shows the blind de-referencing of a pointer to be dangerous. A simple action, such as protecting the de-reference with an exception handler (by using the Windows NT Structured Exception Handling facility, for example), could make a qualitative improvement in reliability.

As a side note, many of those applications that did detect the error did not provide the user with reasonable or pleasant choices. These applications did not follow with an opportunity to save pending changes made to the current document or other open files. Doing a best-effort save of the current work (in a new copy of the user file) might give the user some hope of recovering lost work. Also, none of the applications that we tested saved the user from seeing a dialog pertaining to the cause of the crash that contained the memory address of the instruction that caused the fault, along with a hexadecimal memory dump. To the average application user, this dialog is cryptic and mysterious, and only serves to confuse them.

Our final piece of analysis concerns operating system crashes. Occasionally, during our UNIX study, tests resulted in OS crashes. During this Windows NT study, the operating system remained solid and did not crash as a result of testing. We should note, however, that an early version of the fuzz tool for Windows NT did result in occasional OS crashes. The tool contained a bug that generated mouse events only in the top left corner of the screen. For some reason, these events would occasionally crash Windows NT 4.0, although not in a repeatable fashion.

These results seem to inspire comments such as "Of course! Everyone knows these applications are flaky." But it is important to validate such anecdotal intuitions. These results also provide a concrete basis for comparing applications and for tracking future (we hope) improvements.

Our results also lead to observations about current software testing methodology. While random testing is far from elegant, it does bring to the surface application errors, as evidenced by the numerous crashes encountered during the study. While some of the bugs that produced these crashes may have been low priority for the software makers due to the extreme situations in which

they occur, a simple approach to help find bugs should certainly not be overlooked.

The lack of general access to application source code prevented us from making a more detailed report of the causes of program failures. GNU Emacs and Mozilla were the only applications that we were able to diagnose. This limited diagnosis was useful in that it exposes a trend in poor handling of pointers in event records. In our 1990 and 1995 studies, we were given reasonable access to application source code by the almost all the UNIX vendors. As a result, we provided *bug fixes*, in addition to our bug reports. Today's software market makes this access to application source code more difficult. In some extreme cases (as with database systems, not tested in this study), even the act of reporting bugs or performance data is forbidden by the licence agreements [1] (and the vendors aggressively pursue this restriction). While vendors righteously defend such practices, we believe this works counter to producing reliable systems.

Will the results presented in this paper make a difference? Many of the bugs found in our 1990 UNIX study were still present in 1995. Our 1995 study found that applications based on open source had better reliability than those of the commercial vendors. Following that study, we noted a subsequent overall improvement in software reliability (by our measure). But, as long as vendors and, more importantly, purchasers value features over reliability, our hope for more reliable applications remains muted.

Opportunity for more analysis remains in this project. Our goals include

1. Full testing of the applications on Windows 2000: This goal is not hard to achieve, and we anticipate having the full results shortly.
2. Explanation of the random Win32 message results: We were surprised that the `PostMessage` and `SendMessage` results differed. This difference may be caused by the synchronous vs. asynchronous nature of `PostMessage` and `SendMessage`, or the priority difference between these two types of messages (or other reasons that we have not identified). We are currently exploring the reasons for this difference.
3. Explanation of the Windows NT 4.0 vs. Windows 2000 results: Given that we test identical versions of the applications on Windows NT 4.0 and Windows 2000, our initial guess was that the results would be identical. The differences could be due to several reasons, including timing, size of the screen, or system dependent DLLs. We are currently exploring the reasons for this difference.

6 Related Work

Random testing has been used for many years. In some ways, it is looked upon as primitive by the testing community. In his book on software testing[7], Meyers says that randomly generated input test cases are "at best, an inefficient and ad hoc approach to testing". While the type of testing that we use may be *ad hoc*, we do seem to be able to find bugs in real programs. Our view is that random testing is one tool (and an easy one to use) in a larger software testing toolkit.

An early paper on random testing was published by Duran and Ntafos[3]. In that study, test inputs are chosen at random from a predefined set of test cases. The authors found that random testing fared well when compared to the standard partition testing practice. They were able to track down subtle bugs easily that would otherwise be hard to discover using traditional techniques. They found random testing to be a cost effective testing strategy for many programs, and identified random testing as a mechanism by which to obtain reliability estimates. Our technique is both more primitive and easier to use than the type of random testing used by Duran and Ntafos; we cannot use programmer knowledge to direct the tests, but do not require the construction of test cases.

Two papers have been published by Ghosh *et al* on random black-box testing of applications running on Windows NT[5,6]. These studies are extensions of our earlier 1990 and 1995 Fuzz studies[8,9]. In the NT studies, the authors tested several standard command-line utilities. The Windows NT utilities fared much better their UNIX counterparts, scoring less than 1% failure rate. This study is interesting, but since they only tested a few applications (attrib, chkdsk, comp, expand, fc, find, help, label, and replace) and most commonly used Windows applications are based on graphic interfaces, we felt a need for more extensive testing.

Random testing has also been used to test the UNIX system call interface. The "crashme" utility[2] effectively exercises this interface, and is actively used in Linux kernel developments.

Source Code

The source and binary code for the fuzz tools for Windows NT is available from our Web page at: `ftp://grilled.cs.wisc.edu/fuzz`.

Acknowledgments

We thank Susan Hazlett for her help with running the initial fuzz tests on Windows NT, and John Gardner Jr. for helping with the initial evaluation of the Fuzz NT

tool. We also thank Philip Roth for his careful reading of drafts of this paper. Microsoft helped us in this study by providing a pre-release version of Windows 2000. The paper referees, and especially Jim Gray, provided great feedback during the review process.

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] M. Carey, D. DeWitt, and J. Naughton, "The 007 Benchmark", *1993 ACM SIGMOD International Conference on Management of Data*, May 26-28, 1993, Washington, D.C. pp. 12-21.

[2] G.J. Carrette, "CRASHME: Random Input Testing", `http://people.delphi.com/gjc/crashme.html`, 1996.

[3] J. W. Duran and S.C. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering* **SE-10**, 4, July 1984, pp. 438-444.

[4] S. Garfinkel and G. Spafford, **Practical UNIX & Internet Security**, O'Reilly & Associates, 1996.

[5] A. Ghosh, V. Shah, and M. Schmid, "Testing the Robustness of Windows NT Software", *1998 International Symposium on Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, November 1998.

[6] A. Ghosh, V. Shah, and M. Schmid, "An Approach for Analyzing the Robustness of Windows NT Software", *21st National Information Systems Security Conference*, Crystal City, VA, October 1998.

[7] G. Meyers, **The Art of Software Testing**, Wiley Publishing, New York, 1979.

[8] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", University of Wisconsin-Madison, 1995. Appears (in German translation) as "Empirische Studie zur Zuverlasskeit von UNIX-Utilities: Nichts dazu Gerlernt", *iX*, September 1995. `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisted.ps`.

[9] B. P. Miller, L. Fredriksen, B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM 33*, 12, December 1990, pp. 32-44. Also appears in German translation as "Fatale Fehlerträchtigkeit: Eine Empirische Studie zur Zuverlassigkeit von UNIX-Utilities", *iX* (March 1991). `ftp://grilled.cs.wisc.edu/technical_papers/fuzz.ps`.

[10] C. Petzold, **Programming Windows**, 5th ed., Microsoft Press, Redmond, WA, 1999.

[11] J. Richter, **Advanced Windows**, 3rd ed., Microsoft Press, Redmond, WA, 1997.

[12] D. Solomon, **Inside Windows NT**, 2nd ed., Microsoft Press, Redmond, WA, 1998.

[13] J. A. Whittaker and A. Jorgensen, "Why Software Fails", *Technical Report*, Florida Institute of Technology, 1999, `http://se.fit.edu/papers/SwFails.pdf`.

Gemini Lite: A Non-intrusive Debugger for Windows NT

Ryan S. Wallach
Lucent Technologies

Abstract

It is frequently useful to debug a running software system in a production environment with a symbolic debugger without interfering with the operation of the system. The user of such a debugger may want to inspect data or trigger some data collecting operations whenever the running program hits an arbitrary address. Terminating or initiating the debugging session must also be transparent to users of the system. Debuggers available for Windows NT (or any debugger written with the Win32 debugging API) cannot detach from a running process without killing it, so they are unsuitable for debugging live systems. This paper presents the design of Gemini Lite, a debugger written without the Win32 debugger API, which has the capabilities needed to debug running production systems.

1 Introduction

Lucent Technologies' DEFINITY® Enterprise Communications System is a highly reliable (99.999% uptime) large communications server. The software base contains several million lines of code. Like any large software system, each release of DEFINITY contains software bugs that are not discovered until the system has been installed at a customer's premises. Many bugs are easily reproducible in the development lab from a customer's description. It is impossible, however, to precisely reproduce the conditions of an installed system, and therefore some problems cannot be reproduced in the lab. When this happens, it is necessary to debug the system at the customer's premises without disturbing its operation.

DEFINITY is implemented as a collection of processes in a multitasking proprietary operating system. It contains a proprietary client/server debugger, Gemini, which lets support engineers at a Lucent site securely connect to a customer's system and non-intrusively debug the software. Gemini consists of a small DEFINITY server process (known as the *agent*) which controls the target processes, and a UNIX client process (known as the *host*) that accepts commands and sends messages to the agent to execute them. The host process runs on the support engineer's workstation. It has access to symbolic information for the DEFINITY processes, so it translates symbol names into addresses for the agent.

Gemini supports the traditional model of debugging, that is, the user can set a breakpoint, wait for the process to halt, then single step it and examine data to find the cause of the bug. These features are often used during product development under controlled conditions. However, DEFINITY processes are interrelated and time-dependent. If a process is halted for more than a few milliseconds, the system could (in the course of error recovery) reinitialize itself, and this could disrupt the customer's business. When debugging a live system, then, special debugging capabilities are needed. Gemini provides four key features:

1. Gemini breakpoints can have lists of commands (called action lists) associated with them, and the breakpoints can remain active even when the host isn't running (the agent is always running). When a process hits a breakpoint with an action list, the agent runs the commands, logging any output to an internal buffer, and resumes execution of the debugged process. This mechanism is frequently used to determine where processes are executing and what the relevant data looked like when the breakpoint was hit. Lucent support engineers typically use the host to set breakpoints with action lists and then they exit the debugger and come back hours or days later and view the output buffer to determine if the breakpoint was hit and gather the output from the action list commands.
2. Gemini can debug all the processes in the system in the same session of the debugger. It is even possible to set up breakpoints with action list commands that can manipulate other processes when the first process hits the breakpoint.
3. Gemini is non-intrusive. Users can read and write memory, set and clear breakpoints, and look at the status of DEFINITY processes without halting them. Most of the important data in DEFINITY is global, so developers frequently need to read from (and write to) these data structures without halting the process they're debugging.
4. Gemini can attach to running processes and detach from them cleanly, without interfering with their operation.

In February 2000, Lucent introduced its DEFINITY ONE™ Communications System. DEFINITY ONE contains DEFINITY plus other co-resident applications running on one system under Windows NT 4.0 (all references to Windows NT in this paper refer to this version). DEFINITY ONE required a debugger to run on the platform with the same capabilities as Gemini as well as the ability to debug multithreaded processes. Because no suitable off-the-shelf debugger could be found, we developed our own debugger, Gemini Lite, to provide these capabilities for DEFINITY ONE.

2 The Search for a Debugger on Windows NT

In order to understand the rationale behind Gemini Lite's design, it is important to understand the Win32 Debugging API and how this affects off-the-shelf debuggers for Windows NT.

2.1 The Win32 Debugging API

Windows NT provides an API for developers to create their own debugger [1]. Typically, a debugger attaches to a running process by calling DebugActiveProcess() with the target process id as an argument. This registers the debugger with the operating system. The debugger then calls WaitForDebugEvent() which makes the calling thread of the debugger block until a debugging event is sent to it by the operating system. Windows initially sends events to the debugger to give it handles to each thread in the process being debugged. The debugger then receives events when threads in the target process hit a breakpoint, generate an unhandled exception, etc.[2]

The Win32 debugging API is similar to the ptrace() system call interface used by some UNIX debuggers such as GDB [3] (some versions of UNIX do debugging through the /proc filesystem instead of through ptrace(), and there is nothing similar in Windows NT). To initiate a debug session, a UNIX debugger can call ptrace with a PTRACE_ATTACH request, which allows it to control the target process as if it were its parent. This is analogous to NT's DebugActiveProcess() call. After the debugger has attached to the process, it receives SIGCHLDs when something happens to the target, or it can do a wait() or waitpid() to receive notification of events from the target. The wait() or waitpid() calls are analogous to NT's WaitForDebugEvent() calls.

The substantial difference between the Windows NT and UNIX APIs is that a UNIX debugger can call ptrace() with a PTRACE_DETACH request to disconnect the debugger from the target. The target continues to run after the debugger is disconnected, and the parent-child relationship between the debugger and the target is destroyed. Windows NT does not provide a clean way to detach a target, i.e., there is no call to undo a DebugActiveProcess() request. Furthermore, once the process that has initiated a debugging session exits, Windows NT kills the processes that it was debugging [4]. Microsoft plans to address this issue in a later release of Windows NT (in NT 6.0 or later) [5], but for now there is no workaround.

2.2 Debuggers Investigated

We investigated several Microsoft debuggers for Windows NT (WinDbg, Visual C++ IDE, and ntsd) [6] [7][8] to determine if they could be used for DEFINITY ONE. Each of these appears to use the Win32 API and kills the debugged process when it exits. Since the Microsoft provided debuggers could not satisfy our requirement that they be able to cleanly detach, we investigated commercially available debuggers such as GDB, NuMega's SoftICE [9], and Oasys MULTI [10]. These exhibited the same problem.

Many other debuggers have been built for multithreaded applications on different operating systems [11][12][13], and multi-process, non-intrusive debuggers have also been built [14]. These debuggers have all been built either by using the native debugger API provided by the operating system or extending it to meet the needs of the debugger. Building a debugger (or adopting one of these debuggers) on Windows NT using the debugging API is not acceptable for the reasons discussed above, and since Windows NT is not an open source operating system, it would not be possible to extend it to support one of these debuggers.

GDB is perhaps the most common open-source debugger available, and we considered adapting it. Besides the fact that GDB uses the Win32 debugging API, there were other reasons that we chose not to use it. First, GDB can only debug a single process at a time and is intrusive [15]. This behavior stems from the core of GDB; modifying this would be, says the Cygnus White Paper on GDB, "a daunting task because of its complexities...". Furthermore, our project used the Microsoft Visual C++ 5.0 compiler, and the version of GDB available during our development cycle only supported COFF format symbolic information in the executables. The Microsoft compiler only emits CodeView symbolic information in executable files (and DLLs).

Because no suitable off-the-shelf debugger could be found, we developed Gemini Lite. Gemini Lite is a general-purpose Windows NT debugger. It can be used to debug any NT process (not just DEFINITY) assuming the process is properly linked. Gemini Lite is

non-intrusive and does not use the Win32 debugging API. It has the basic features of other debuggers, but its architecture permits it to have unattended action lists and to debug processes without killing them once it exits.

3 Design of Gemini Lite

3.1 Overview

The Win32 API provides the basic mechanisms to implement basic debugging features in Gemini Lite. Table 1 shows which Win32 functions can be used to implement the core features of the debugger [16].

Feature	Win32 API Used
Read/Write Memory, set/clear breakpoints	ReadProcessMemory(), WriteProcessMemory()
Read/Write Registers, control single stepping	GetThreadContext(), SetThreadContext()
Halt/Resume a thread	SuspendThread(), ResumeThread()
Determine when a thread hits a breakpoint, steps, or has some other exception	Structured Exception Handling mechanisms

Table 1. Win32 support for debugger features

Win32 calls that refer to the target's memory require a handle to the target process, which can be obtained from a call to OpenProcess(). Similarly, calls that refer to threads (e.g., SuspendThread()), require a handle to the target thread, which can only be obtained by that thread (or the thread which created it) [17] because there is no OpenThread() call in Windows NT 4.0 (Microsoft has added this to the Win32 API in Windows 2000). Using structured exception handling mechanisms for breakpoints presents a similar problem; a process cannot change the exception mechanisms of threads in other processes.

When DebugActiveProcess() is used to implement a debugger, Windows NT sends the debugger the handles to the desired threads. Without using this API, the only way for the debugger to have access to the thread handles is for it (or the part of it that actually controls other processes) to be integrated into the application code. Due to the size of the DEFINITY code base, it was not feasible to change the application code to accommodate the debugger. We used a client/server approach to separate the portion of Gemini Lite that interacts with the user from the portion that controls processes, which must be somehow linked into the application.

The first part of Gemini Lite, the debugger process, is what the user runs to access the debugger. It acts like DEFINITY's Gemini host, accepting input from the user and sending the input to the server to be parsed and executed. The server part of the architecture, which is the core of Gemini Lite, is a DLL that is linked with the applications that can be debugged (for the rest of this paper, "the DLL" refers to this). The DLL takes the place of the Gemini agent and is responsible for parsing and executing the commands sent by the debugger process. The DLL must be linked with both the debugger process and all the target processes.

To force the application processes to link with the DLL without changing their code, they must be linked (with the Visual C++ linker) using the **–include <symbol>** directive and the appropriate export library for the DLL. The **–include** directive places a reference to the specified symbol (which is some globally exported symbol in the DLL) into the executable, which forces the DLL to be loaded when the process is run [18]. This limits the utility of the debugger somewhat, as it can only debug processes that are linked with the DLL, but for DEFINITY ONE this was an acceptable constraint.

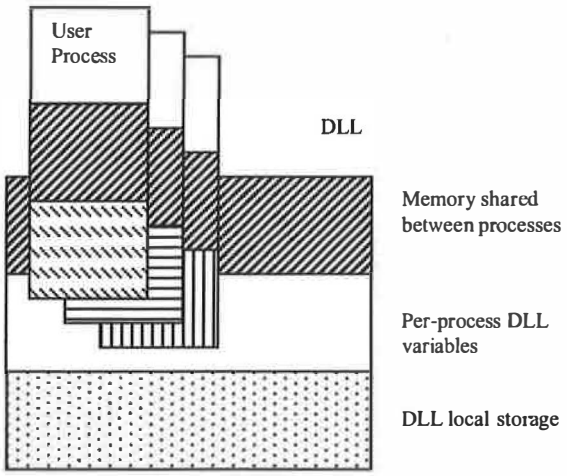


Figure 1. Overview of Gemini Lite's Architecture

A general overview Gemini Lite's architecture appears in Figure 1. The figure illustrates that all processes linked with the DLL share some common memory. The contents of the shared memory are defined by the

DLL; it contains exported functions as well as shared data. The DLL defines another set of exported variables; this set of variables has unique copies in each of the debugged processes. The DLL also has non-exported code and data that are not visible to them.

Windows NT forces each process linked with the DLL to call the DllMain() function in the DLL when the process and its threads are created and when they exit [19]. The Gemini Lite DLL uses this property to obtain handles to each debuggable thread (i.e., a thread in a process linked with the DLL) by having the thread store a handle to itself in the shared memory area of the DLL. DllMain also has each thread set its unhandled exception filter to a routine inside the DLL. This exception filter is used to catch breakpoint and single step exceptions, which would typically not be caught by the application. All other exceptions that are caught by this handler are sent back to Windows NT to resolve, although they could just as easily be handled by the debugger and reported to the user.

The debugger itself is just another user process that can call routines inside the DLL that implement its functions. These routines are exported to all processes, not just the debugger, and this architecture makes it possible for processes to call debugger routines when they hit breakpoints, which will be discussed in detail below.

3.2 Shared Memory in the DLL

The shared memory area of the DLL contains data that must be shared between the debugger and the user processes. The data structures must be statically allocated at compile time because any memory dynamically allocated by one process would not be in the address space of other processes, including the debugger. Furthermore, there is no guarantee that the DLL will be mapped to the same address space in each user process, so any traditional data structure that uses pointers (e.g., linked lists or hash tables) is not suitable for the DLL. Array-based hash tables, lists, and queue template classes were defined to hold the shared data.

The data structures contained in the shared memory area include:

1. Information about each process registered with the DLL (e.g., pid and creation time)
2. Information about each thread registered (e.g., thread id, handle to the thread, state of the thread)
3. Information about each breakpoint set (e.g., process in which it's set, address, commands to run when hit)
4. A queue that contains messages generated from functions in the DLL to be displayed by the debugger.

These data structures are protected by mutexes to ensure correctness.

The shared memory area also contains variables that track whether the debugger is running. Since the debugger is a user process, it also calls DllMain() when it starts. DllMain() checks the name of each calling process, and when it finds the name of the debugger process (a predefined name), it notes that the debugger is running. Another mechanism could also be used in DllMain() to determine which user process is the debugger. The DLL needs to know whether the debugger is attached because certain status messages (e.g., breakpoints being hit) are written to a queue in the shared memory area by functions in the DLL. The debugger has a thread that looks at this queue and displays the messages to the user.

3.3 Process and Thread Registration in the DLL

As mentioned before, Windows NT forces each process and thread linked with the DLL to call DllMain() when they are created. NT passes a parameter to DllMain() that indicates the reason for the call. When new processes start up, they call DllMain() one or more times. A process's first call to DllMain() has this parameter set to DLL_PROCESS_ATTACH. This notifies the DLL that the process (and its primary thread) has attached to the DLL. Subsequent calls by threads in the process to DllMain() set the parameter to DLL_THREAD_ATTACH and inform the DLL that additional threads in the process have been created.

When the Gemini Lite DLL's DllMain() is called with a DLL_PROCESS_ATTACH message, the DLL determines whether the process is the debugger or an application process. As mentioned above, if the process is the debugger, the DLL stores its pid in shared memory and sets a status variable in the DLL to reflect that the debugger is active. For the primary thread (and other threads) of user processes, the DLL creates an object to represent the thread in its shared memory area. The thread id and handle are stored in the object. DllMain() then sets the thread's unhandled exception filter to point to a routine inside the DLL.

Processes and threads also notify the DLL when they exit normally. When a thread exits, NT forces it to call DllMain() with a reason of DLL_THREAD_DETACH. Similarly, when a process exits, NT forces it to call DllMain() with a reason of DLL_PROCESS_DETACH. Note that the call with DLL_THREAD_DETACH is not made for all threads that are running when the process exits; only the DLL_PROCESS_DETACH call is made. When the

DLL gets these calls, it frees the object and associated data structures that were allocated for the thread (or threads, if the process detached) including its breakpoints.

In some circumstances (e.g., a call to TerminateProcess() or TerminateThread()), it is possible that processes and threads can be terminated without calling DllMain(). When this happens, the DLL does not know that the process or thread is gone, so it cannot free the related data structures. Because the tables holding the data are statically allocated and were sized to accommodate the number of threads and processes running in DEFINITY ONE, it is possible that they may fill up with information about processes and threads that no longer exist.

If the tables are full when the DLL attempts to register a process, the DLL checks all registered threads to make sure that they are still valid, and it frees up entries that are no longer valid.

Because the debugger is just another user process, the DLL can detect when it exits through its call to DllMain(). In order to prevent any application processes that have breakpoints set from stopping, the DLL disables all breakpoints that may have been set in other processes and it resumes execution of any threads that may have been stopped when it detects that the debugger exited.

3.4 Debugger Process

The Gemini Lite debugger process has two threads. The main thread runs in a loop which prompts the user for commands, reads the command line, and calls the appropriate functions in the DLL to parse and execute the command. The second thread repeatedly locks the mutex protecting the message queue in the DLL shared memory, removes and displays any messages found in the queue, then releases the mutex. As a result, the user is immediately informed of events such as breakpoints being hit regardless of what he or she may be doing in the debugger (typing commands or viewing output).

4 Implementation of Debugging Features

4.1 Symbolic Debugging

Debuggers like GDB typically read symbolic information for the process they are debugging from the executable file and build internal symbol tables for use by the debugger. Gemini Lite does not directly read the symbolic information for the processes and threads that it debugs. Instead, it relies on the Win32 symbol handling routines contained in IMAGEHLP.DLL [20]. These routines provide the capability to obtain an address in a running process from the name of a global symbol and vice-versa. The first time the user issues a command for a thread in a process that takes an address as a parameter, Gemini Lite calls SymInitialize() and passes it the handle to the process to initialize the symbol handler. It then loads the symbols for the process by enumerating all its modules and calling SymLoadModule() for each of them. Once the symbols have been loaded, Gemini Lite uses SymGetSymFromName() to translate global symbol names into an address or SymGetSymFromAddress() to translate an address into a global name.

In Windows NT 4.0, IMAGEHLP.DLL does not provide facilities for translating a file name and line number into an address and vice versa. Microsoft has added the SymGetLineFromAddr() and SymGetLineFromName() functions to the Win32 API in Windows 2000 to accomplish this. In order to perform this function in Windows NT 4.0, a program would have to directly examine the CodeView debugging information in the executables (or in separate .DBG files). Time constraints only permitted us to display file and line number information in Gemini Lite's disassembly routines. Other Gemini Lite commands (such as for setting breakpoints) cannot accept a file and line number in place of a text address.

Use of the IMAGEHLP.DLL symbol handling functions requires that the DLLs and EXEs that make up the processes being debugged are compiled with debugging information. The debugging information must be compiled into the objects, not placed in a program database (PDB) file. However, it is usually undesirable to ship production code without stripping debugging information. To avoid this, we used the **rebase** tool shipped with Visual C++ to strip the debugging information from the compiled objects and place it in separate .DBG files. When the application needs to be debugged, the .DBG files are copied to the target machine, and then the _NT_SYMBOL_PATH environment variable is set before running Gemini Lite. This environment variable tells the IMAGEHLP.DLL symbol handling routines where to find the symbols. We ship the symbol files (in an encrypted form) with the DEFINITY ONE system. When support engineers need to debug, they use Windows RAS or a TCP/IP network to establish a connection to the system. They then decrypt the symbol files and run the debugger in a window directly on the target system.

4.2 Stopping and Restarting Execution

The debugger can force threads to halt execution by calling SuspendThread() with the handle to the thread. The debugger obtains the handle from the shared memory area in the DLL. Before using the handle, the

debugger must call DuplicateHandle() to obtain a handle in its context; the handle stored in the DLL is a handle in the context of the process that registered with the DLL. To resume execution of a thread, the debugger calls ResumeThread(), passing it the handle to the thread.

4.3 Reading and Writing Memory

The debugger commands that need to read or write memory do so by calling ReadProcessMemory() and WriteProcessMemory(). The debugger must have permission to read the memory of the processes it's debugging. When the debugger is debugging processes started by the same user, this is not a problem. For DEFINITY ONE, we require that the debugger can be started only by a privileged user. Some processes we need to debug are started by a system service. Ordinarily, a user process does not have permission to access a system service. We modified the default discretionary access control lists (DACLs) of our system level processes to give the account that can run the debugger full access to them.

4.4 Reading and Writing Registers

Registers in a thread can only be read by reading the thread context. Debugger commands that need to read registers first use SuspendThread() to stop the thread unless it is already halted. They then call GetThreadContext() to retrieve the context of the thread, which includes the contents of the registers. After the context is obtained, ResumeThread() is called if the thread needs to continue execution.

To write to a register, the context image returned by GetThreadContext() is modified to contain the updated register value, then SetThreadContext() is used to write the modified context back to the thread.

4.5 Breakpoints

Like most debuggers for software running on x86 processors, Gemini Lite sets breakpoints in a process by replacing the first byte of the instruction at the breakpoint address with 0xcc (INT3). The original instruction byte is saved in the record for the breakpoint in the shared memory area of the DLL so that it can be restored later. Because all threads in a process have the same address space, a breakpoint set in a process will affect all the threads in the process.

Figure 2 illustrates the sequence of events that occurs when a thread hits a breakpoint. First, the thread raises a breakpoint exception when it executes the instruction at the breakpoint address. The system stores the thread's context in a context record (the value of the EIP register in the record is set to the address where the thread encountered the exception) and forces the thread to call the appropriate exception filter. If no other exception filter handles breakpoint exceptions (which is a requirement for processes linked with the DLL), then the exception filter in the DLL (which was set as the unhandled exception for the process when it attached to the DLL) will be called. The exception filter receives a pointer to the context record as well as a pointer to an exception record, which contains the exception code, the address at which the exception occurred, and other information.

In the exception filter in the DLL, the thread first examines the exception record to determine the cause of the exception. If the type is not EXCEPTION_BREAKPOINT or EXCEPTION_SINGLE_STEP, then the exception filter will return EXCEPTION_CONTINUE_SEARCH, which will force NT to handle the exception. This will either terminate the process or invoke the system debugger, depending on the system's registry settings.

If the exception type is EXCEPTION_BREAKPOINT, then the thread checks the list of breakpoints in shared memory of the DLL to determine if a breakpoint was set at the exception address. If no breakpoint is found, there is no way for the thread to continue, so the filter will return EXCEPTION_CONTINUE_SEARCH.

If a breakpoint is found, the thread determines if it should halt. Breakpoints may have a threshold stored in the object representing them that specifies the number of times the breakpoint is to be hit before a thread will stop. Also, the thread will only halt if the debugger is running, as indicated by the variable that the debugger sets when it registers with the DLL.

If the thread determines that it must halt, it creates a message notifying the user that the breakpoint has been hit, and it puts it in the message queue for the debugger. It sets a variable in its record in the DLL's shared memory indicating that it is suspended due to the breakpoint, and then it suspends itself by calling SuspendThread() with its thread handle as a parameter. The debugger process that the user is running, meanwhile, contains two threads. One reads and executes commands from the user, and the other checks the message queue from the DLL. After the target thread puts the message into the queue indicating that it hit the breakpoint, this thread of the debugger displays it to the user.

When the user decides to resume execution of the thread, he or she gives the appropriate command to the debugger, which calls a function in the DLL. This function examines the record for the thread in shared memory. If the state of the thread indicates that it has

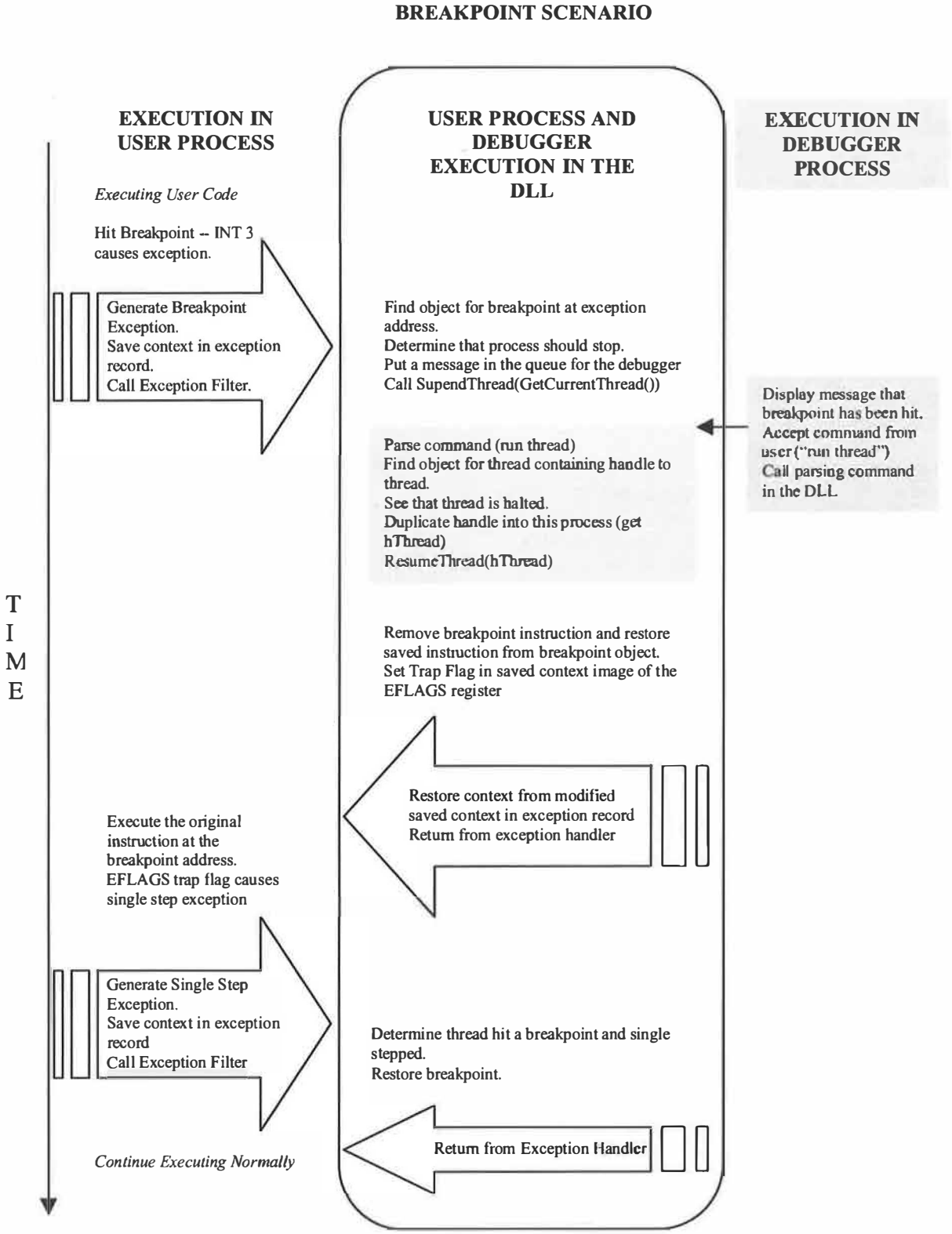


Figure 2. View of execution when a thread hits a breakpoint.

been halted at a breakpoint, then the function gets the handle to the thread that is stored in shared memory and passes it to ResumeThread(). This wakes up the thread that hit the breakpoint.

When the thread wakes up it is still in the exception handler, and the breakpoint instruction is still at the breakpoint address. The thread replaces the breakpoint instruction with the original instruction, sets status variables in its record in shared memory to indicate that it has been resumed after a breakpoint, and sets the Trap Flag in the image of the EFLAGS register stored in the saved thread context that was passed to the exception handler. It then returns EXCEPTION_CONTINUE_EXECUTION. This forces NT to restore its context from the saved image (with the modified EFLAGS register) and continue executing where the exception occurred.

The thread executes the real instruction at the breakpoint address, and then, because the Trap Flag in the EFLAGS register is set, it generates a single step exception. Again, Windows NT forces the thread to call the unhandled exception filter. In the exception filter, the thread sees that the exception code is EXCEPTION_SINGLE_STEP. It checks the status variables in shared memory and figures out that it has single stepped after the previously hit breakpoint. The thread then saves the instruction at the breakpoint address, reinserts the breakpoint, and again the exception filter returns EXCEPTION_CONTINUE_EXECUTION, which lets the thread continue executing at the instruction after the breakpoint. The thread then continues executing until some other event occurs.

Gemini Lite's handling of breakpoints differs from the traditional implementation. In a debugger written using the Win32 API, for example, when a thread hit a breakpoint, the *system* would suspend it. A debugger doing a WaitForDebugEvent() would be woken up, and it would decide whether to keep the process halted or restart it with a call to ContinueDebugEvent() (after replacing the breakpoint instruction with the original instruction and setting EFLAGS appropriately). In Gemini Lite, the *thread* decides itself whether it should be suspended, and it suspends itself. In both cases, the debugger causes the thread to resume execution. In the Win32 case, the thread resumes where it was stopped by the system, in the application code. In Gemini Lite, the thread resumes execution in the exception handler. When it returns from the handler, the system causes it to resume executing where the exception was raised.

4.6 *Action Lists*

An action list is a list of debugger commands to be executed when a breakpoint is hit. When the Gemini Lite user sets a breakpoint in a thread, he or she may also supply the action list. The action list is stored with the breakpoint information in the shared memory area in the DLL. In the exception filter, if the thread determines that a breakpoint has an action list, instead of calling SuspendThread(), it reads the list of action list commands from shared memory and passes them to the same function in the DLL that the debugger executable runs to execute commands that are input by the user. Since all the functions of Gemini Lite are also in the DLL, they can be executed just as if the user were giving them on the command line. The output from the commands is directed to a large circular buffer, also in shared memory. The output will stay in the buffer until the Gemini Lite user clears it. Note that if a breakpoint has an action list, Gemini Lite does not have to be running in order for the action list commands to execute, because the thread automatically resumes execution after the action list commands are run. This makes action lists very useful for unattended debugging. The user can set up the breakpoints with action list commands to dump data of interest, exit Gemini Lite, and come back later to examine the data.

The user can also set a flag in the DLL's shared memory area that the thread will check after it executes the action list commands. If the flag is set and the debugger is running, the thread will generate a message for the debugger that tells the user that the breakpoint was hit. This feature can be used in combination with an empty action list to let the user know that the thread executed code at a particular instruction without having to halt the thread.

4.7 *Single-Stepping*

When a thread is stopped, either after being halted by the user or by hitting a breakpoint, the user may wish to step through the execution of the program being debugged. Because Gemini Lite relies only on the IMAGEHLP.DLL symbol handler to read debugging information, it does not have access to the information that links an address to the program source file and line number. Consequently, Gemini Lite can only step through a program any number of assembly-language instructions at a time.

The implementation of single stepping was seen above in the discussion of breakpoints. When the target thread is halted, the user's single step command sets the Trap Flag in the EFLAGS register (by getting the thread context, modifying, and writing it back, if the thread is not halted after a breakpoint, or by modifying

the saved context in the exception record, if it is), and resumes execution of the thread (by calling ResumeThread()). The user can specify the number of instructions to single-step; this number is stored in shared memory in the DLL.

After resuming execution, the target thread executes one instruction and generates an exception, sending it into the exception filter in the DLL. If the thread has stepped the desired number of instructions, it puts a message in the queue for the debugger to inform the user that it halted, then it calls SuspendThread() on itself. Otherwise, the thread decrements the step count, returns from the exception filter, and continues stepping.

After the thread is finally halted, the user can resume execution of the thread or single-step it again. As with breakpoints, when the thread is in the exception filter it checks to see if Gemini Lite is running before calling SuspendThread(). If the debugger is not present, the thread will not stop. This avoids the situation where a user requests a single step of a large number of instructions, but then exits the debugger before the stepping is completed.

5 Related Tools

Since the IMAGEHLP.DLL routines only locate global symbol names, we needed a set of tools to use with Gemini Lite which could show us the layout of structures in memory, addresses of individual array elements, and addresses of global functions and variables. In the UNIX environment, these functions are provided by tools like objdump (from GNU) and nm. On Windows NT, the Microsoft provided tools to do these things (such as dumpbin) are part of Visual C++ and cannot be run without it. We developed a standalone set of tools to do these things. The development was difficult, in part, because Microsoft's compilers emit symbolic information in a proprietary format (CodeView), and Microsoft does not provide any libraries for manipulating this information. We generated our own set of routines from Microsoft's symbolic debugging information specification [21].

6 Conclusion

Gemini Lite was used during the development of DEFINITY ONE to solve some difficult problems. In one case, an uninitialized variable was causing incorrect information to be displayed on DEFINITY's administration terminal. We set breakpoints with empty action lists both where we knew the code had executed and where we thought it should be executing. When these breakpoints are hit, Gemini Lite puts a message into its output buffer. By looking at the buffer, we were able to see where the code failed to branch as we thought it should. At that point, we used an action list to display a variable that determined where the code branched. After seeing that the value in this variable could not have been set by the code that had executed, a close examination of the code showed that the variable had not been initialized.

Our experience with Gemini Lite suggests some enhancements. First, Gemini Lite could be enhanced to read CodeView information from the processes it's debugging and maintain its own symbol table. With this information, Gemini Lite would have a knowledge of variable type information, mapping of source files and line numbers to addresses, locations and names of local variables in functions, and more information that would enable it to be a source level debugger instead of an assembly level debugger. A networked client-server approach to Gemini Lite has also been proposed which would eliminate the need to keep the symbol files (.DBG files) on the system being debugged.

7 Acknowledgements

I would like to thank Bhavesh Davda, Bill Lyford, and David Walters for their assistance during the development of Gemini Lite.

8 References

[1] "Platform SDK: Debugging and Error Handling: Debugging Reference: Debugging Functions", *MSDN Library,* Microsoft Corp., October, 1999.

[2] Kath, Randy. "The Win32 Debugging Application Programming Interface", Microsoft Corp., November 5, 1992.

[3] Stallman, Richard and Pesch, Roland. *Debugging with GDB, the GNU Source-Level Debugger, Fifth Edition*, Free Software Foundation, April, 1998.

[4] "PRB: Debugee Exits When the Debugger Exits, ID Q164205", *Microsoft Knowledge Base,* Microsoft Corp., February 28, 1997.

[5] Private Conversations with Microsoft Premier Support Representatives.

[6] "Platform SDK: Tools: Symbolic Debuggers", *MSDN Library*, Microsoft Corp., October, 1999.

[7] "Platform SDK: Tools: WinDbg", *MSDN Library,* Microsoft Corp., October, 1999.

[8] "Visual C++ User's Guide: Debugger", *MSDN Library,* Microsoft Corp., October, 1999.

[9] SoftICE is a trademark of NuMega Technologies, Inc.

[10] MULTI is a trademark of Green Hills Software and XEL, Inc.

[11] Buhr, Peter, Karsten, Martin, and Shih, Jun. "KDB: A Multi-threaded Debugger for Multi-threaded Applications". Proceedings of the

SIGMETRICS Symposium on Parallel and Distributed Tools, ACM, 1996, pp. 80 – 87.
[12] Caswell, Deborah, and Black, David. "Implementing a Mach Debugger for Multithreaded Applications". Proceedings of the Winter 1990 USENIX Conference.
[13] Redell, David. "Experience with Topaz TeleDebugging". Digital Equipment Corporation, Systems Research Center.
[14] Himelstein, Mark and Rowell, Peter. "Multi-process Debugging". USENIX Conference Proceedings, Summer, 1985.
[15] Shebs, Stan. "GDB: An Open Source Debugger for Embedded Development", Cygnus Support White Paper available at http://www.redhat.com/support/wpapers/cygnus_gdb.
[16] See the appropriate page for each function in *MSDN Library*, Microsoft Corp., October, 1999.
[17] "Platform SDK: DLLs, Processes, and Threads: Thread Handles and Identifiers", *MSDN Library*, Microsoft Corp., October, 1999.
[18] "Visual C++ Programmer's Guide: Compiler Reference", *MSDN Library,* Microsoft Corp., October, 1999.
[19] Sarma, Debabrata. "DLLs for Beginners", Microsoft Developer Support, November, 1996.
[20] Pietrek, Matt. "Under the Hood", *Microsoft Systems Journal,* May, 1997.
[21] Smith, Steve, and Spalding, Dan, et al. "Visual C++ Symbolic Debug Information Specification, Revision 5, 32-Bit only September, 30,1996", *MSDN Library*, Microsoft Corp., October, 1999.

Extending the Windows Desktop Interface With Connected Handheld Computers

Brad A. Myers, Robert C. Miller, Benjamin Bostwick, and Carl Evankovich

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu
http://www.cs.cmu.edu/~pebbles

ABSTRACT

Increasingly, people will be in situations where there are multiple communicating computing devices that have input / output capabilities. For example, people might carry their handheld computer, such as a Windows CE device or a Palm Pilot, into a room with a desktop or an embedded computer. The handheld computer can communicate with the PC using wired or wireless technologies, and then both computers can be used at the same time. We are investigating many ways in which the handheld computer can be used to *extend* the functions of existing and new applications. For example, the handheld's screen can be used as a customizable input and output device, to provide various controls for desktop applications. The handheld can contain scroll bars, buttons, virtual knobs and menus. It can also display the slide notes or a list of slide titles for a presentation, the list of current tasks and windows, and lists of links for web browsing. The user can tap on these lists on the handheld to control the PC. Information and control can flow fluidly among all the user's devices, so that there is an integrated environment.

1. INTRODUCTION

The age of *ubiquitous computing* [20] is at hand with computing devices of different shapes and sizes appearing in offices, homes, classrooms, and in people's pockets. Many environments contain embedded computers and data projectors, including offices, classrooms (e.g., [1]), meeting rooms, and even homes. One little-studied aspect of these environments is how personal, handheld computers will interoperate with the desktop computers. More and more people are carrying around programmable computers in the form of Personal Digital Assistants (PDAs) such as a Palm Pilot or Windows CE device, and even cell-phones and watches are becoming programmable. We are researching the question of what users can do with their handheld computers in such an environment.[1]

Some researchers have looked at using handheld devices in group settings to support collaborative work, usually with custom applications [1] [6] [13] [15] [17] [18] [19]. However, there has been little study of how portable devices can work in conjunction with the Windows desktop user interface, and with conventional Windows applications. Most of the research and development about handheld computers has focused on how they can be used to *replace* a regular computer for when one is not available. The conventional model for PDAs is that the data is "synchronized" with a PC once a day using the supplied cradle, and otherwise the PDA works independently. This will soon change. CMU has installed a Lucent Wavelan wireless network (which implements the IEEE 802.11 wireless standard) throughout the campus, in a project called "Wireless Andrew" [9]. Many Windows CE handheld computers can be connected to this wireless network using a PCMCIA card (although this is not an ideal solution since 802.11 has a high-demand for power and drains the batteries quickly). This year, the BlueTooth standard for small device wireless radio communication [8] will finally be available, and most PDAs, cell-phones, and other computerized small devices are expected to support it. Therefore, we expect

[1] This paper uses the terms PDAs and handheld computers interchangeably, since most of the ideas in this paper would equally apply to any of these devices. Our research has so far focused on using PDAs such as Palm Pilots and Windows CE devices. We will use "PC" to refer to the "regular" computer, which might be a desktop or laptop computer, or a computer embedded in a room with a large wall-mounted display.

that connecting the PCs and handhelds together will no longer be an occasional event for synchronization. Instead, the devices will frequently be in close, interactive communication. We are studying how the handheld computer's display, input mechanisms, and processor can be used to enhance the desktop computer's application when they are communicating. For example, the handheld can provide extra views of the data on the PC, and there can be buttons on the handheld that control the PC's applications. Thus, rather that trying to repeat the user's desktop and desktop applications on the handheld, we use the handheld to augment and enhance the existing PC. We feel that some of these applications may be sufficiently interesting to warrant buying a PDA (for example, the Slideshow Commander discussed below), whereas others would mainly provide small incremental benefits to people who already own a PDA. This paper presents an overview of the applications we have created for individual use of handhelds and PCs together.

2. OVERVIEW

In the "old days," computers had a variety of input switches and knobs. Today, computers are standardized with a keyboard and a mouse for input, and connecting other input devices can be difficult and expensive. Although today's computers have high-resolution screens and window managers, users can still need extra space to display information. For people who have already purchased a PDA, we want to exploit it to provide the benefits of having an extra input and output device. For example, Figure 1 shows the PDA being used at the same time as the mouse as an extra input and output device for the non-dominant hand. Note that we are not claiming that a PDA is an "ideal" extra input device, or that it is necessarily better than special-purpose extra devices that have been created (e.g., [2] [3]). Rather, we observe that people do not buy such devices, but they do have a PDA, so maybe we can get some of the benefits of the special-purpose devices without accruing the costs.

When the PDA is connected to the PC, we have found that the PDA can be used to *extend* the desktop user interface in various ways. It can serve as a customizable input device, with "soft" buttons, sliders, menus and other controls displayed on the screen. These can be made big enough to operate with a finger, even with the non-dominant hand (as in Figure 1). We have shown that using the non-dominant hand this way is effective for certain interaction techniques, such as scrolling [12]. The PDA can also be used as an output device to provide secondary views. This is useful when the entire PC screen is engaged and unavailable. For example, during a PowerPoint presentation, the PDA can display the notes of the current slide. Another use is to display information that should not be covered by other windows, such as the Windows' task bar, without sacrificing desktop screen real estate. Since the interfaces are on a handheld, they can be carried with the user, and even used with different PCs. Although some of the applications discussed below might work as a separate window on the main PC screen, the advantages of using a separate device are that the user can operate the PDA screen with a finger, it can be used at the same time as a mouse (supporting two-handed input), it can provide additional screen space when the PC's screen is full, and it provides an interface that users can take with them to use with different computers.

Figure 1. A PDA in its cradle on the left of the keyboard, and a person using both the PDA and the mouse simultaneously.

3. APPROACH

In order to support a variety of mobile devices, each running various programs, all attached to a PC running many different applications, we provide a central dispatcher we call PebblesPC (a later section presents the architecture in more detail). Our various PDA applications communicate with PebblesPC using a serial cable, infrared, Lucent's Wavelan radio network, Symbol's Spectrum24 radio network, or BlueTooth [8]. The appropriate PC-side program is run, which interprets the messages from the PDA. Many of the applications discussed below interface with conventional Windows applications, such as Microsoft Office. In other cases, we have created new, custom applications, for example to support multi-user drawing (these are discussed elsewhere [13]).

By interfacing with existing PC applications, we can explore how handheld computers can extend conventional Windows user interfaces, and how the handheld might integrate with the user's existing information environment. The goals of the Pebbles programs include providing mobile remote control of the PC application, moving displays and controls from the PC screen onto the handheld, to free up the PC screen

for other tasks, and to enable two-handed operation of applications, for example so the left hand can be used for scrolling. To investigate and demonstrate these issues, we have created a wide range of applications. The following sections discuss some of the applications we have created.

3.1 Slideshow Commander

Often when giving presentations from a computer using slide show programs such as Microsoft's PowerPoint, the speaker will use notes. These are usually printed on paper, which makes them difficult to change. The speaker usually clicks on the mouse or keyboard to move forward and backward through the slides, which means that the speaker must be close to the PC. In a study of PowerPoint presentations [5], a number of problems were identified, including that the speaker often desires to walk away from the presentation computer to be closer to the audience (and some people just like to wander around while talking). It can also be awkward to point to and annotate the slides using a mouse. Further, people often had trouble when trying to navigate to the previous slide or to a particular slide.

The Slideshow Commander application (Figure 2) solves these problems by moving some of the display and control functions to the PDA. On the PC side, it works with Microsoft PowerPoint 97 or 2000 running under Windows. The PDA side works with any Windows CE machine or a Palm Pilot in black-and-white or color. Pressing the physical buttons on the Palm Pilot, using the scroll wheel on the side of a Windows CE palm-size device (Figure 2b), tapping the on-screen buttons with the stylus, or else giving a Graffiti or Jot gesture using a finger or the stylus, causes the slides to advance or go backwards. There are various panels on the PDA that support other tasks during a presentation. The Scribble panel allows the user to point and scribble on a thumbnail picture of the current slide on the PDA (see the top of Figure 2a and 2c), and have the same marks appear on the main screen. For the thumbnail picture to be legible on gray-scale PDAs, we have PowerPoint generate a "black-and-white" picture of the slide, which removes the background.

The Notes panel (Figure 2a on the bottom, and 2d) displays the notes associated with the current slide. The notes page is updated whenever the slide is changed, so it always displays the notes for the current slide. PowerPoint allows each slide to have associated notes that some people use—especially when the slides consist mostly of pictures. Other people put most of their text on the slides as bullet points. The Notes view supports both styles of presentations. The user can choose whether to display the text that appears on the slide itself, the text from the notes associated with that slide, or both.

The Titles panel (see Figure 2b) displays the titles of all the slides in the current talk. The currently displayed slide is highlighted, and tapping on a slide name changes the presentation to that slide. This might be useful for people who have a large slide set and want to dynamically choose which slides to use for a given talk. Another use is at the end of the talk, during questions, to jump back to a specific slide under discussion. Finally, the Timer panel (Figure 2e) displays a timer, which can count up or down or display the current time of day. This is useful for timing the talk.

Of all the Pebbles applications, the Slideshow Commander is probably the most popular. We have received many positive comments on it, and the free version (which does not have the thumbnail pictures) is in wide scale use. The new version is being released commercially by Synergy Solutions (www.synsolutions.com).

3.2 Scrollers

Research [3] [12] [21] has shown that people can effectively scroll documents using their non-dominant hand instead of using conventional scroll bars. Recently, there has been a profusion of devices to help with scrolling, including the Microsoft IntelliMouse and the IBM ScrollPoint mouse. These mechanisms all use the dominant hand. Pebble's scrolling applications allow the PDA to be used as a scrolling device in either hand. Figure 3 shows some of the scrollers we have created. A user study demonstrated that these could match or beat scrolling using the mouse with conventional scroll bars, and was significantly faster than other scrolling mechanisms such as the scroll wheel built into mice [12]. As part of the same study, we measured how long it takes a person to move from the keyboard to acquire input devices, and found that the penalty for moving both hands (the left hand to the PDA and the right hand to the mouse) is only about 15% slower than moving just the right hand to the mouse and leaving the left hand on the keyboard. The movement time to return to the keyboard was only 13% slower. Thus, using the PDA does not provide a significant penalty, and can increase the speed of scrolling without requiring special-purpose devices.

Although these applications were designed to support scrolling, they can also be used as an extra, parallel input stream from the mouse. For example, in a posi-

tioning and resizing task [3], the scroll events can be mapped to change the object's size, which will support two-handed operation. In the future, we will be experimenting with other tasks that require more than two degrees of freedom, such as translation and rotation in 3-D worlds.

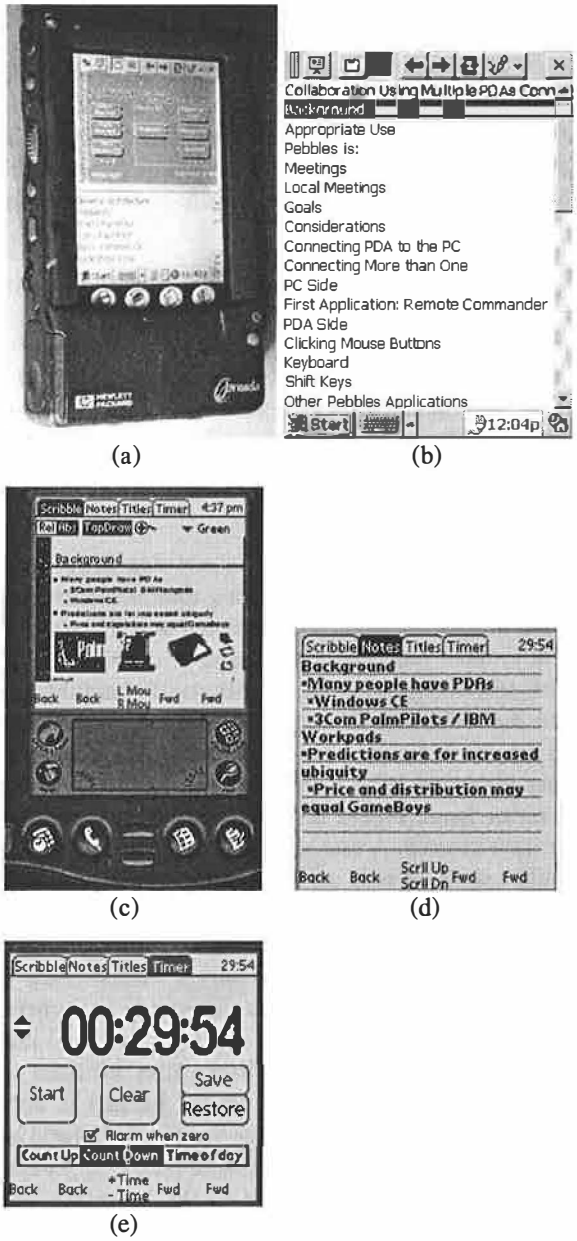


Figure 2. The Pebbles Slideshow Commander program. (a) The Scribble and Notes panes running in Windows CE on a HP Jornada. (b) The Title panel under Windows CE. (c) The full Palm Pilot, with the "Scribble" panel at the front. The "Notes" (d) and "Timer" (e) panels. Meanwhile, a PC is running PowerPoint and a PDA is in continuous two-way communication with the PC.

3.3 Task and Window Switcher

The Windows desktop provides a "Taskbar" at the bottom of the screen to switch among applications. Additionally, many Multi-Document Interface (MDI) applications have a "Windows" menu item to switch among windows in that application. The Switcher application (see Figure 4a) combines both capabilities into a single user interface on the PDA. This eliminates the confusion of which of these very different mechanisms to use to get to a desired window, and provides a consistent, quick, and always-available mechanism that does not use up any valuable real estate on the main screen. The user can tap on an item on the PDA to cause that window to come to the front of the PC. Another advantage is that we can optionally group windows by application, even for multiple instances of the application. Since Switcher provides various ways of organizing the application's windows, this can make it easier to find a window. For example, if there are multiple instances of Notepad running, all the documents from all of them can be combined into one alphabetical list, rather than by the order they were opened.

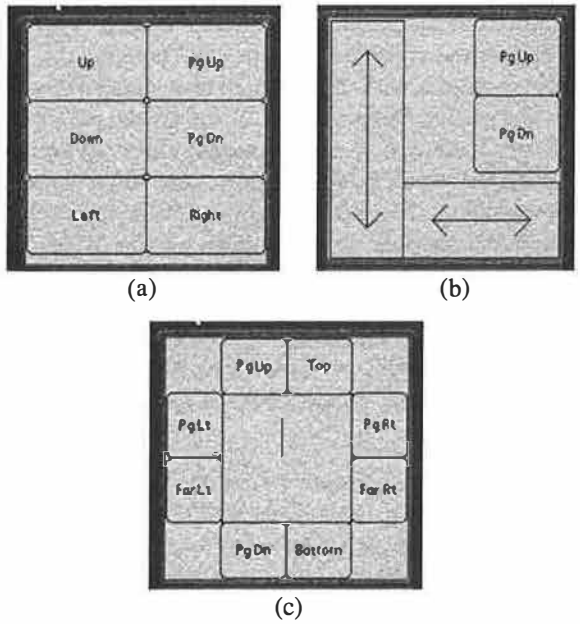


Figure 3. Three scrolling interfaces: (a) Buttons that autorepeat to scroll up and down a line or a page, or left and right. (b) A slider, where dragging a finger or stylus in a rectangle drags the text the same amount. (c) A virtual rate-

controlled joystick, where pressing in the center and moving out scrolls the text in that direction at a rate proportional to the distance.

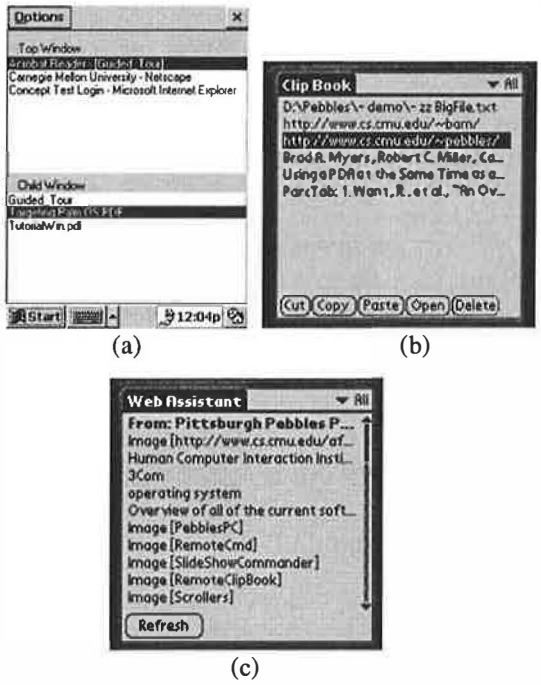


Figure 4. (a) Task Switcher displays on the PDA the top-level tasks and the windows for the selected task. (b) Remote Clipboard connects the PC's clipboard and the PDA's. Here is a list of items pasted from the PC, which can be copied back to the PC or to another PDA application. The "Open" button causes the selected file or URL to be opened on the PC. (c) Web Assistant displays the links on the current page. Clicking on a link causes the browser to go to that page. The list of links is only updated when the Refresh button is hit. For images that are links, the "alt" text is displayed in square brackets next to the word "Image" but if there is no "alt" text, then the URL is displayed instead.

3.4 Remote Clipboard

PDAs generally have some sort of synchronization mechanism to keep the information on the PDA consistent with a PC. However, the process is somewhat inconvenient and time consuming. For example, on the Palm Pilot you need to HotSync, which copies all applications' data that have changed. This is inappropriate if you want to just transfer a small amount of data, especially if it is to someone else's PC, to whom one does not want to give all of one's personal data. To provide a quicker way to copy small amounts of data between the PDA and the PC, we developed Remote Clipboard [10], which connects together the clipboards of the PC and the PDA. Whenever the user copies or cuts text on either machine, it can be pasted on the other. Using this familiar interaction technique, users can easily transfer information among many kinds of machines, devices and applications.

File names or URLs can also be pasted onto the PDA, and the Open button on the PDA will cause the PC to open the application associated with the file, or open the web page in the default browser. This allows information on the PC to be copied to the PDA either by value or by reference. If the user copies the information itself and pastes it to the PDA, this corresponds to passing the information "by value." If the user copies the filename or URL of the information, then it is passed "by reference." The Remote Clipboard PDA application (see Figure 4b) provides one place to store the data on the PDA, but the information can be pasted and copied from any PDA application, including the address book, scheduler, MemoPad, etc. In the future, we will investigate transferring richer information than just text. For example, it would be convenient to quickly sketch ideas on the PDA, and paste the pictures onto the PC to use as notes or as templates for more careful drawings.

3.5 Web Assistant

Research [4] and experience show that web browsing often takes on a "hub and spoke" style, where the user frequently returns to a main index page in order to find the next out-link to click on. Examples include the results of a search, and table of contents and maps of web sites and on-line documents. The Web Assistant application aids in these tasks by allowing links from the "hub" page to be copied to the PDA.

Figure 4c shows a view of the list of links on the PDA. Clicking on any link causes the web browser on the PC to switch to the specified page. Note that this does *not* automatically refill the PDA display with the links from the new page—it continues to show the original set of links so the user can easily move from link to link. Pressing the Refresh button refills the PDA page with the current page's links.

Many web pages are filled with irrelevant links. For example, a search results page can easily have more advertisement links than results. Therefore, the user can select a region of the browser text and only copy the links from that region onto the PDA. In the future, we expect to integrate more intelligent parsing technology [11] into the Web Assistant so the useful links can be selected and copied more automatically.

3.6 Shortcutter

The Shortcutter application combines many of the features of the previous utilities to allow users to cre-

ate custom panels of "shortcut" buttons, sliders, knobs, and pads to control any PC application. The buttons can be big enough to hit with a finger, or tiny so that many will fit on a screen. The Shortcutter can provide customizable interfaces on the PDA even for applications that do not have a customization facility on the PC. Since these are on the PDA, you can take them with you and use them even on other people's computers. In edit mode, users can draw panels and assign an action to each item in the panel. Switching to run mode, the items will perform their actions.

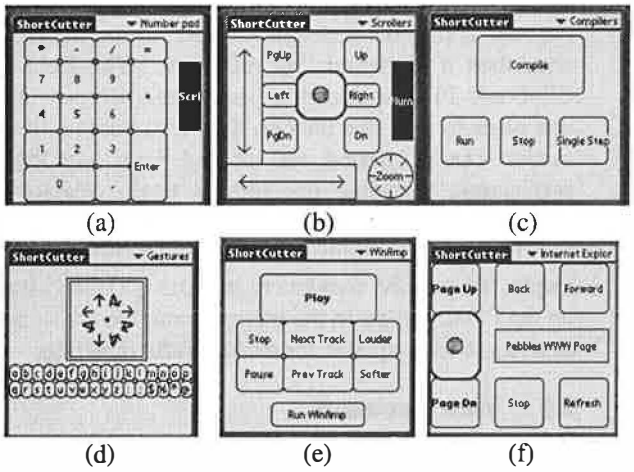


Figure 5. Panels created with Shortcutter. (a) A numeric keypad, (b) a collection of scrollers and a knob, (c) buttons for controlling any of a set of compilers, (d) a gesture pad and two rows of small buttons, (e) a controller for the WinAmp PC program, and (f) a panel for browsing in Internet Explorer.

We have defined several kinds of widgets that can be added to panels:

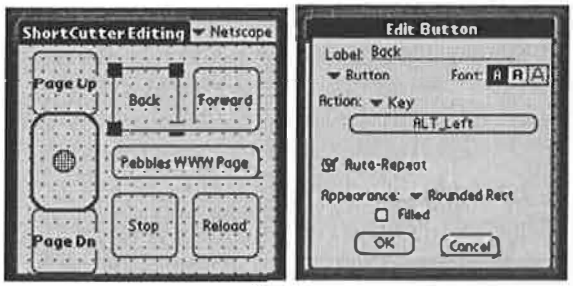
- **Buttons:** The user can assign a label, and use various shapes and patterns. Buttons can be specified to auto-repeat if the user presses and holds on them, or to only execute once per tap. Pressing a button invokes one of the actions listed below.
- **Scrolling Widget:** The vertical and horizontal sliders and the rate-control joystick scroller (see Figure 3) are also available in the Shortcutter (see Figure 5b which includes two sliders, a rate-control joystick in the center, and some scroll buttons). Scrolling is implemented on the PC by sending WM_HSCROLL and WM_VSCROLL messages to the foreground window.
- **Mouse Pad:** This supplies a rectangular area within which the user drags the stylus to perform mouse movements. It works like a trackpad on some laptops. Mouse events are inserted in the Windows event queue with the mouse_event API.
- **Gesture Pad:** This item allows users to give gestures such as those used in previous studies [2]. Figure 5d shows the gesture pad with icons for the nine supported gestures: tap (dot), strokes in four directions, and back-and-forth strokes in four directions. Each of these gestures can be assigned a different action. In the future, we might support more elaborate gestures or even user-trainable gestures.
- **Knobs:** For knob widgets, the user can specify one action to be generated when the user makes a clockwise gesture and another for counter-clockwise. These actions are repeatedly sent as the gesture continues, to give the effect of "turning" the knob. For example, a keyboard key that increments a value (such as the "+" key in Quicken, or a string like "+1=" for a calculator) might be assigned to one direction and decrementing to the other direction. Since this is a general facility, any action can be used. Figure 5b shows a knob for zooming.

For many of these widgets, the user can choose any of the following actions:

- **Key:** Send any PC keyboard key to the PC as if the user had typed it, including function and other special keys and modifiers such as Control and Alt. This action is implemented by inserting keystrokes in the Windows event queue with the keybd_event API. This action makes it easy to create a panel like a numeric keypad (Figure 5a) that might be useful with laptops that do not have one. Anything that can be invoked using keyboard keys (menu items, etc.) can be easily assigned to a Shortcutter button.
- **String:** Send a string to the PC as if the user had typed it on the keyboard. This action is implemented by inserting each character of the string as if it were a keystroke. This might be useful for creating buttons that serve as abbreviations, or as input to the PC during a macro.
- **Open File/URL**: To tell the PC to open a file in its application, or to go to a specified page in a browser. This action is implemented by passing filename or URL to the Windows ShellExecute API.
- **Run Application:** Causes an application to be run, given an executable filename and optional command-line arguments. The user can specify whether to run a new instance of the application (using the WinExec API) or switch to the application if it is already running. The user can specify the executable filename of an application, possibly by finding the executable for the application, or a shortcut to it possibly in the Start Menu or on the desktop. An-

other, often more convenient way is to ask Shortcutter to determine the executable filename of a running application. On Windows platforms that support the Toolhelp32 API (Windows 95 and Windows 2000), Shortcutter finds the executable by calling CreateToolhelp32Snapshot to get information about the running process. On other platforms (specifically NT 4), Shortcutter injects itself in the address space of the running process with a window hook in order to call the GetModuleFilename API. Shortcutter uses this same technique to switch to a running application, by enumerating windows and searching for a window created by the given executable filename.

- **Scrolling:** Causes the foreground window to scroll horizontally or vertically, by lines or pages. This action is implemented by sending WM_HSCROLL and WM_VSCROLL messages.
- **Switch Panel:** The user can create a button that goes to a particular panel. This does not send anything to the PC. The black buttons on the right of Figures 5a and 5b switch between these two panels.
- **Mouse Button**: This is for sending mouse button events, such as left or right buttons down and double-clicking. Parameters include the modifier keys, if any (to support events like SHIFT-CLICK). Mouse buttons are simulated by calling the Windows mouse_event API.
- **Recorded Event:** This action is created by recording the next WM_COMMAND message sent by a menu item or toolbar button, using a global windows hook. When the action is invoked later, it replays the message to the current foreground window. This allows items on the PDA to perform actions that may not have a keyboard equivalent in the application, such as changing modes in a drawing program. Some menus and toolbars cannot be recorded in this fashion, however, either because they do not use WM_COMMAND messages, or because the messages must be sent to a window other than the main application window.
- **Macro:** This allows a sequence of the above actions to be associated with an item. Note that these macros can operate across multiple PC applications. Macros can invoke other macros, which supports procedural abstraction.
- **Wait for User:** This pops up a window with a button that waits for the user to tap OK. It is only really useful in macros. A string can be displayed on the PDA to instruct the user what is expected. A cancel button aborts the operation of the macro.
- **Application-Specific:** An item can have multiple actions associated with it, where each action is specific to a different PC application. Then, if the button is pressed, Shortcutter checks which PC application is in the foreground (using the techniques described above under Run Application), and chooses the appropriate action. This can provide a uniform, virtual interface to a set of applications that perform the same function but have disparate interfaces. An example use for this is that we use a variety of programming environments that unfortunately have different key assignments for actions and are not customizable. We created a panel of application-specific buttons that send the appropriate key for the current environment (see Figure 5c).



(a) (b)

Figure 6. (a) The editing mode of Shortcutter with the Back button selected. (b) Setting the properties of the Back button so when you click on the Back button at run-time, it sends to the PC the keystroke Alt-left-arrow.

The panels are constructed on the PDA by switching into "Edit Mode" (see Figure 6a). This supplies a conventional direct-manipulation editor, where buttons can be simply drawn, and their labels and properties assigned. The property sheet for a button is shown in Figure 6b. For the more complicated actions, other forms are used.

To make Shortcutter even more useful we allow the hardware (physical) buttons on the PDA to be mapped to any action. Panels and items can be stored on the PC and reused, and useful ones can be easily shared. Users can design buttons that are large enough to be hit with a finger (as in Figures 4 and 5), or very small so many items can fit on a panel and can be invoked using the stylus (as in the bottom of Figure 5d). The result is a very flexible and useful application with which users can create many interesting personalized shortcuts that might make their everyday use of a PC more effective.

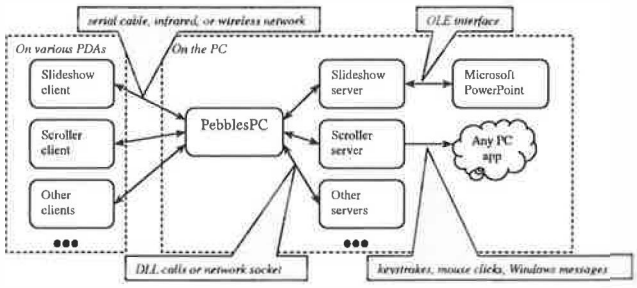


Figure 7. The Pebbles architecture.

4. ARCHITECTURE

The general architecture of Pebbles is shown in Figure 7. The main components are client programs running on (one or more) PDAs, server programs running on the PC, and PebblesPC, a PC program that mediates between clients and servers. These components communicate using a flexible message protocol we have designed.

4.1 Clients

Client programs run on handheld devices. Most of our applications run on both Palm and Windows CE devices. Multiple handhelds can be connected to the same PC, enabling not only multi-user applications but also the single-user, multi-device applications described in this paper. We generally assume that a handheld device can run only one program at a time. (This assumption is always true on the Palm, but on Windows CE a program can continue to run in the background, although you cannot see it.) Thus we make no allowances for multiplexing a serial connection between multiple simultaneously active client programs. The user is free to switch among client programs at any time, however.

4.2 Servers

Server programs run on the Windows PC. Our architecture supports two kinds of servers. The first kind uses plugins, which are dynamic link libraries (DLLs) loaded into PebblesPC's address space. Each plugin runs in its own thread to avoid blocking PebblesPC, and each plugin thread has a private Windows message queue. Windows messages are used to communicate between the plugin and PebblesPC. The second kind of server is a separate process, running either on the same PC or a remote host, and communicating with PebblesPC through a network socket.

Servers perform their operation in various ways, with various levels of application independence. For example, the Slideshow Commander server interacts directly with PowerPoint through OLE Automation. This kind of server clearly requires significant knowledge of the application being controlled. At the other extreme, the Scroller server simulates scrolling by inserting keystrokes and Windows messages into the standard Windows event stream. This kind of server need not know anything about the Windows applications that eventually receive the input events.

4.3 PebblesPC

PebblesPC acts as both a naming service and a message router. A server makes itself available to clients by connecting to PebblesPC and registering its name. For plugin servers, this happens automatically when PebblesPC loads the plugin's DLL, and the server's name is derived from the DLL filename. Clients connect to a server by first connecting to PebblesPC and requesting a server name (such as "Slideshow Commander"). If a server by that name is available, then PebblesPC makes a virtual connection between the client and the server, routing messages back and forth. PebblesPC allows clients and servers to connect through heterogeneous I/O interfaces, including serial ports, infrared, network sockets, and Windows message passing. PebblesPC handles the low-level details of each interface.

4.4 Message Protocol

Clients and servers communicate using an asynchronous message protocol designed to be simple, lightweight, and easy to implement. Low overhead is vital because many Pebbles applications use the handheld as a pointing device, which sends frequent update messages over a low-bandwidth channel (such as a serial port). Each message consists of a 1-byte command field (indicating the type of message), a 2-byte length field (extensible to 4 bytes if necessary), and a data field. Several command values are reserved for PebblesPC functions, such as registering client and server names, requesting a connection to a server, and closing a connection. Messages with other command values are passed unchanged between client and server, allowing the command to have an arbitrary meaning. For example, the Slideshow Commander application uses various command codes for requesting slides, changing the current slide, and sending slide titles, text, and thumbnail images. These messages can be sent directly through the serial cable or infrared, or else the messages can be sent over a TCP stream for network connections.

We have developed libraries for the Palm Pilot, Windows CE and Windows operating systems that implement the Pebbles protocol for various I/O interfaces, including serial, infrared, network sockets, and

Windows messages. This makes creating new Pebbles applications relatively easy.

5. RELATED WORK

The Xerox ParcTab [19] project investigated using small handheld devices at the same time as other computers. The ParcTab was used to investigate some aspects of remote cursors and informal voting about how well the speaker was doing. These applications are more closely related to the groupware Pebbles applications discussed in our previous paper [13]. One application allowed ParcTab to be scripted in Tcl/Tk to create applications that performed such functions as moving forward and backwards in a Tcl/Tk presentation manager, controlling the stylus to move the PCs cursor, remotely controlling a camera, creating postit notes on a PC, controlling Mosaic, and switching among X11 windows on Unix [15].

Rekimoto has studied a number of ways to use a PDA along with other computers. For example, Pick-And-Drop [16] and HyperDragging [18] are new ways to move information among different devices, which are related to (but different from) our cut-and-paste multi-machine model. The M-Pad system [17] supports multiple users collaborating with PDAs and a large whiteboard, which is similar to our multi-user application [13]. The large body of work on multi-modal user interfaces (e.g., [14]) typically combines direct manipulation with gestures and speech, but does not typically discuss the use of a hand-held device with a PC.

There have been many investigations of using both hands at the same time to control a computer. For example, an early study [3] showed that people could effectively scroll and change objects' size with the left hand while positioning with the right hand, and that many people operated both devices at the same time. Another study investigated how accurately gestures could be drawn with the non-dominant hand on a small touchpad mounted on top of a mouse [2]. Cross-application macros, such as provided by Shortcutter have been available in other applications. However, Shortcutter adds several new kinds of actions to the script, and the ability to keep the scripts on a PDA that can be easily transported.

Other groups are studying the use of Palm Pilots in various settings, where they are *not* connected to a PC. For example, NotePals synchronizes people's independent notes after a meeting [6], and Georgia Tech's "Classroom 2000" project [1] is studying the use of handheld computers in classrooms. For PDAs connected to a PC, our previous paper [13] discusses the use of multiple PDAs connected to a PC to support meetings. After the release of an earlier version our of Slideshow Commander for the presenter, a different group created a system with similar capabilities aimed at helping the audience members who have a PDA [7]. The current work shows that a PDA can be equally useful for a single person as an extension to the Windows user interface for desktop applications, and discusses the various ways we have extended the PC desktop user interface using the PDA.

6. STATUS AND FUTURE WORK

All of the applications discussed here are available for downloading off the World-Wide Web at http://www.cs.cmu.edu/~pebbles. These applications have been downloaded over 15,000 times. The newest version of Slideshow Commander is available commercially from Synergy Solutions (www.synsolutions.com). Third parties are also picking up on our architecture. For example, a commercial company, Iron Creek Software, used our architecture to build a Palm interface to the popular WinAmp PC program for playing MP3 and other digital music. Figure 5e shows a Shortcutter WinAmp controller, but the Iron Creek version also supports downloading and rearranging play-lists on the PDA.

Current work in our project is focusing on two areas: mixing private and public information, and classroom use. In many public meetings, it is useful for individuals to privately get on their handheld more details about publicly displayed information on a wall display. Alternatively, users might have additional details on their handheld that they want to combine with the public record. We are exploring how to make this information flow be fluid and natural. Another important area of work will be on transitioning some of these applications to the classroom. For example, the Slideshow Commander might broadcast to all the audience's computers the thumbnail picture and the notes of the current slide, to facilitate notetaking and understanding.

In general, we will continue to explore the many ways that PDAs can be used at the same time as desktop computers to enhance the user's effectiveness. Surprisingly, this is an area that has received very little study. As the communication mechanisms improve, and handheld computers become more capable and ubiquitous, it will be increasingly important to consider how the users' mobile devices can interoperate with the stationary computers in the environment. The research described here presents a number of ways in which the conventional Windows user interface can be extended using mobile devices.

ACKNOWLEDGMENTS

For help with this paper, we would like to thank Andrew Faulring, Brad Vander Zanden, Karen Cross, Rich McDaniel, Bernita Myers and the referees.

This research is supported by grants from DARPA, Microsoft, Symbol, HP, IBM and Palm. This research was performed in part in connection with Contract number DAAD17-99-C-0061 with the U.S. Army Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U.S. Army Research Laboratory or the U.S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

REFERENCES

1. Abowd, G.D., *et al.* "Investigating the capture, integration and access problem of ubiquitous computing in an educational setting," in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 440-447.

2. Balakrishnan, R. and Patel, P. "The PadMouse: Facilitating Selection and Spatial Positioning for the Non-Dominant Hand," in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 9-16.

3. Buxton, W. and Myers, B. "A Study in Two-Handed Input," in *Proceedings SIGCHI'86: Human Factors in Computing Systems.* 1986. Boston, MA: pp. 321-326.

4. Card, S.K., Robertson, G.G., and York, W. "The WebBook and the Web-Forager: An Information Workspace for the World-Wide Web," in *Proceedings CHI'96: Human Factors in Computing Systems.* 1996. Vancouver, BC, Canada: pp. 111-117.

5. Cross, K. and Warmack, A. "Contextual Inquiry: Quantification and Use in Videotaped Analysis," in *Adjunct Proceedings CHI'2000: Human Factors in Computing Systems.* 2000. The Hague, The Netherlands: pp. To appear.

6. Davis, R.C., *et al.* "NotePals: Lightweight Note Sharing by the Group, for the Group," in *Proceedings, CHI'99: Human Factors in Computing Systems.* 1999. Pittsburgh, PA: ACM. pp. 338-345.

7. Dey, A.K., *et al.* "The Conference Assistant: Combining Context-Awareness with Wearable Computing," in *Proceedings of the 3rd International Symposium on Wearable Computers.* 1999. San Francisco, CA: pp. To Appear.

8. Haartsen, J., *et al.*, "Bluetooth: Vision, Goals, and Architecture." *ACM Mobile Computing and Communications Review*, 1998. **2**(4): pp. 38-45. Oct. www.bluetooth.com.

9. Hills, A., "Wireless Andrew." *IEEE Spectrum*, 1999. **36**(6)June.

10. Miller, R.C. and Myers, B. "Synchronizing Clipboards of Multiple Computers," in *Proceedings UIST'99: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1999. Asheville, NC: pp. 65-66.

11. Miller, R.C. and Myers, B.A. "Lightweight Structured Text Processing," in *Usenix Annual Technical Conference.* 1999. Monterey, California: pp. 131-144.

12. Myers, B.A., Lie, K.P.L., and Yang, B.-C.J. "Two-Handed Input Using a PDA And a Mouse," in *Proceedings CHI'2000: Human Factors in Computing Systems.* 2000. The Hague, The Netherlands: pp. To Appear.

13. Myers, B.A., Stiel, H., and Gargiulo, R. "Collaboration Using Multiple PDAs Connected to a PC," in *Proceedings CSCW'98: ACM Conference on Computer-Supported Cooperative Work.* 1998. Seattle, WA: pp. 285-294.

14. Oviatt, S. and Cohen, P., "Multimodal Interfaces That Process What Comes Naturally." *Communications of the ACM*, 2000. **43**(3): pp. 45-53. March.

15. Petersen, K. "Tcl/Tk for a Digital Personal Assistant," in *Also Xerox PARC CSL Technical Report: CSL-94-16, Dec. 1994.: Proc. of the USENIX Symp. on Very High Level Languages (VHLL).* 1994. Santa Fe, New Mexico: pp. 41--56.

16. Rekimoto, J. "Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments," in *Proceedings UIST'97: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1997. Banff, Alberta, Canada: pp. 31-39.

17. Rekimoto, J. "A Multiple Device Approach for Supporting Whiteboard-based Interactions," in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 344-351.

18. Rekimoto, J. and Saitoh, M. "Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments," in *Proceedings SIGCHI'99: Human Factors in Computing Systems.* 1999. Pittsburgh, PA: pp. 378-385.

19. Want, R., *et al.*, "An Overview of the ParcTab Ubiquitous Computing Experiment." *IEEE Personal Communications*, 1995. : pp. 28-43. December. Also appears as Xerox PARC Technical Report CSL-95-1, March, 1995.

20. Weiser, M., "Some Computer Science Issues in Ubiquitous Computing." *CACM*, 1993. **36**(7): pp. 74-83. July.

21. Zhai, S., Smith, B.A., and Selker, T. "Improving Browsing Performance: A Study of Four Input Devices for Scrolling and Pointing," in *Proceedings of Interact97: The Sixth IFIP Conference on Human-Computer Interaction.* 1997. Sydney, Australia: pp. 286-292.

Opportunities for Bandwidth Adaptation in Microsoft Office Documents

Eyal de Lara[†], Dan S. Wallach[‡], and Willy Zwaenepoel[‡]

[†] Department of Electrical and Computer Engineering
[‡] Department of Computer Science
Rice University
{delara,dwallach,willy}@cs.rice.edu

Abstract

Microsoft Office, the most popular office productivity suite, produces large documents that can result in long download latencies for platforms with limited bandwidth. To reduce latency and improve the user's experience, these documents need to be adapted for transmission on a limited-bandwidth network.

To identify opportunities for adaptation, we characterize documents created by three popular applications from the Microsoft Office suite: Word, PowerPoint, and Excel. Our study encompasses over 12,500 documents retrieved from 935 different Web sites.

Our main conclusions are: 1) Microsoft Office documents are large and require adaptation on bandwidth-limited clients; 2) embedded objects and images account for the majority of the data in these documents, with image types being the most popular non-text content, suggesting that adaptation efforts should focus on these elements; 3) compression considerably reduces the size of these documents; and 4) the internal structure of these documents (pages, slides, or sheets) can be used to download elements on demand and reduce user-perceived latency.

1 Introduction

Microsoft Office is the most popular productivity suite for creating documents. Its popularity derives, to some extent, from its ability to create *compound documents* that include data from more than one application. The potentially large size of these documents results in long download and upload latencies for mobile clients accessing the documents through bandwidth-limited links [3, 11, 22]. To reduce latency and improve the user's experience, compound documents and the applications that operate on them need to adapt to the available bandwidth.

To identify opportunities for adapting compound documents we need to understand their main characteristics. However, most studies of content types, especially those done on the Web [4, 21, 23, 24] have consistently ignored compound documents or treated them as opaque data streams, ignoring the rich internal structure that can be used to enhance bandwidth adaptation. In this paper we present an analysis of Office compound documents downloaded from the Web. We focus on those characteristics of Office documents that have implications for bandwidth-limited clients, and identify opportunities for adaptation. Although we report our findings with an emphasis on bandwidth-limited clients, we believe that these results will be useful for office suite designers and people interested in working with compound documents in general.

We undertook this study as part of our Puppeteer project, which uses component-based technology to adapt applications for different operating environments. Puppeteer is well suited for adapting compound documents that include data generated by several software components. By exposing the hierarchy of component data in the compound document, and making calls to the run-time APIs that the components expose, Puppeteer adapts applications without changing their source code. In contrast, traditional adaptation approaches have not been successful for applications that operate on compound documents mainly because the complex

and proprietary nature of these applications thwarts source code modifications [9, 10] and the inclusion of several complex data types, usually embedded in a single file, makes system-based adaptation hard [12, 18, 20].

For this paper, we studied compound documents generated by three popular applications of the Office suite: Word, PowerPoint, and Excel. We chose to focus on Office applications based on four factors. First, Office is the most widely-used productivity suite. Moreover, a significant number of Microsoft Office documents are available on the Web, enabling us to gather the data for our experiments. Second, the Office file formats, although proprietary, are reasonably well documented. Third, the Office applications are highly integrated with each other and have published run-time APIs that can be used by Puppeteer to adapt the applications. Fourth, Office 2000 supports two native file formats: the proprietary OLE-based binary format and a new XML format. By using Office 2000 to convert old files to the new XML format, we can compare the tradeoffs of using a proprietary binary-based file format against a modern standards-based text format, both as intermediate formats suitable for document editing, and as publishing formats, suitable only for reading.

Although we concentrate exclusively on Office documents, we believe that our results apply to compound documents generated by other productivity suites. Since most of these suites support roughly the same features (embedding, images, etc), and document content is driven largely by user needs, it is likely that the main characteristics of documents produced by various productivity suites (*e.g.* distribution of document size, percentage that have images, number of pages, slides, etc.) would be similar.

We downloaded over 12,500 documents, comprising over 4 GB of data, from 935 different sites. Our main results are:

1. Office documents are large, with average sizes of 196 KB, 891 KB, and 115 KB for Word, PowerPoint and Excel respectively. Their large sizes suggest a need for adaptation in low bandwidth situations.

2. Office documents are component rich. 18.19% of Word documents and 46.38% of PowerPoint documents have at least one embedded component. Images were the most common component type.

3. In large documents, images and components account for the majority of the data, suggesting that they should be the main target of the adaptation effort.

4. For small documents, the XML format produces much larger documents than OLE. For large documents, there is little difference.

5. Compression considerably reduces the size of documents in both formats. Moreover, once compressed there is no significant difference in the sizes of the two file formats.

6. XML formats are easier to parse and manipulate than the OLE-binary formats.

The rest of this document is organized as follows. Section 2 provides some background on compound documents and their enabling technology. We also discuss relevant characteristics of the three Office applications that we use in this study. Section 3 describes the documents we used in our experiments. Section 4 presents our experimental results. Section 5 discusses the relevance of our findings to other productivity suites. Finally, section 6 discusses our conclusions.

2 Background

To its user, a compound document appears to be a single unit of information, but in fact it can contain elements created by different applications. A compound document could, for instance, consist of a spreadsheet and several images embedded into a text document.

In the general case, every data type in a compound document (spreadsheet, text, image, sound, etc.) is created and managed by a different application. The different applications used to create the document can be thought of as *software components* that provide services that are invoked to create, edit, and display the compound document.

In the remainder of this section we review the technologies used by Office to enable compound documents. We start with an overview of COM, OLE,

and Automation. We then talk about the two native file formats supported by Office. Finally, we present a taxonomy of components found in Office applications.

2.1 COM, OLE and Automation

Office compound documents are based on the Component Object Model (COM) [5] and the Object Linking and Embedding (OLE) [6] standards, which govern the interactions between the various software components used to create compound documents.

COM enables software components to export well-defined interfaces and interact with one another. In COM, software components implement their services as one or more COM objects. Every object implements one or more interfaces, each of which exports a number of methods. COM components communicate by invoking these methods.

OLE is a set of standard COM interfaces that enable users to create compound document by *linking* and *embedding* objects (components) into container applications, hence the name OLE.

Automation is an OLE technology, which enables third party applications to remotely control Office applications. Puppeteer adapts applications, to a large extent, by invoking Automation interfaces to modify application behavior when executing on bandwidth limited platforms. For example, using Automation interfaces, Puppeteer can adapt a large PowerPoint presentation by loading only a couple of slides, instead of the full presentation, before returning control to the user. While the user works on these slides, Puppeteer loads the remaining slides in the background, and as new slides become available, it instructs PowerPoint to append them to the presentation.

2.2 File formats

Office 2000 supports two native file formats: the traditional OLE-based binary format (hereafter, "OLE archive") and a new XML-based format. The OLE archives [13, 14, 15] rely on the OLE Structured Storage Interface (SSI) to provide a unified view of the compound document in a single file. SSI implements an abstraction similar to a file system within a single file. It supports two types of objects: *storages* and *streams*. Storages are analogous to directories and contain streams or more storages. Streams are analogous to files and contain the components' data. Office applications vary in the way they use the OLE SSI to store embedded objects. Word and Excel, for instance, use a separate storage for every embedded component, making the component structure of the document visible to the OLE SSI. For example, figure 1 shows the structure of a Word archive with two embedded components. Notice how Word keeps each embedded object in a separate SSI storage. In contrast, PowerPoint compresses embedded object native data and stores it in the main application stream. While this strategy increases document compression, it limits the ability of third-party applications to manipulate components within a PowerPoint document.

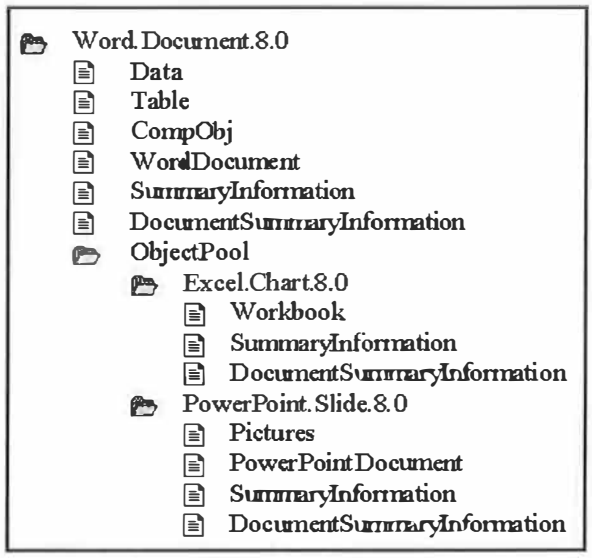


Figure 1: Word archive. The figure shows two embedded objects, an Excel chart and a PowerPoint slide, each stored in a separate SSI storage.

The new XML format [17] provides a more browser-friendly option for storing Office documents. While an OLE archive appears as a single file, an XML document appears as an entire directory of XML files, approximately one per component, image, or slide. The current implementation of Office supports two forms of XML output: a compact low-fidelity representation that can be read by browsers but cannot be edited by Office tools, and a larger high quality representation that supports editing. In this study we focus on the latter XML representation because it is semantically comparable to the

OLE archive.

Aside from the number of files that they use, the two file formats differ mostly in their representation of text and formating information. Images and embedded component native data have similar representation in both formats, with the caveat that component data in the XML-based format is stored in a compressed OLE archive. Moreover, both formats keep in persistent storage two versions of the OLE components they embed. The first one consists of the embedded component's native data, which is used to initialize the component. This data is created and managed by the component itself. The second representation is a cached image of the state of the component the last time it was instantiated. This image, although created by the component, is managed by the container application. This image serves two purposes. First, it allows the document to be rendered quickly, since the code that understands the component's specific type need not be executed until the user wishes to modify the component. Second, the cached image allows the document to be rendered even on systems where some component types are not installed.

There is a significant difference in the way Office supports these two file formats. Office is able to load OLE archives incrementally over a random access file system. In contrast, XML documents must be read in their entirety before control is returned to the user, leading to higher latencies for opening and storing XML-based documents.

2.3 Component taxonomy

Conceptually, Office documents may have up to three classes of components: images, OLE-based embedded components, and virtual components. Images are graphic data that are stored and manipulated directly by the application. This includes the cached versions of any embedded components and any graphic data that the application manipulates directly. OLE-based embedded components are data created using a separate application, as described above. Among the most common types of embedded components are components that implement image types. To differentiate these image types (which are created by a separate application, and hence a type of component) from the primitive images managed by the application we will use the term "image components." Finally, virtual components are objects that are not implemented as OLE-based components but that are perceived by the user as separate entities (*i.e.*, pages in Word, slides in PowerPoint, and sheets in Excel).

Application	Documents	Sites
Word	6481	236
PowerPoint	2167	334
Excel	4056	378

Table 1: Data set. This table presents for each application, the number of documents and the number of Web sites from which they originated.

3 Data set

We collected Word, PowerPoint, and Excel documents from the Web. First, we used the AltaVista search engine [1] to obtain an initial set of URLs. In the first two weeks of October 1999, we searched for pages having links to files with suffixes we were interested in (`doc`, `ppt`, and `xls`). For example, we used the query *link:ppt domain:edu* to search for HTML pages in the edu domain that have links to PowerPoint documents. Then, we used GNU Wget [19] to recursively retrieve documents from our initial search results.

The reliance on a search engine to obtain the documents raises the question of the set representativity. On one hand, a search engine is likely to produce results that are dependent on the popularity of certain pages and documents, skewing the distribution towards these particular document types and producing a non-random set of documents. On the other hand we observe that our documents are fairly well distributed among domains, covering a wide range of user types. Moreover, the shape of the document size plots of section 4.1 and their close fit to the power-law distribution are similar to the results obtained by Cunha *et. al.* [7] in a study of client-based traces covering over half a million user requests for WWW documents.

All downloaded documents were in the binary OLE archive format. Because Office file formats vary from one version of Office to another, we first converted all our data to the Office 2000 formats. We removed documents that appeared to be corrupt or were not actually Office documents. The `doc` suffix, in particular, tends to be used by many appli-

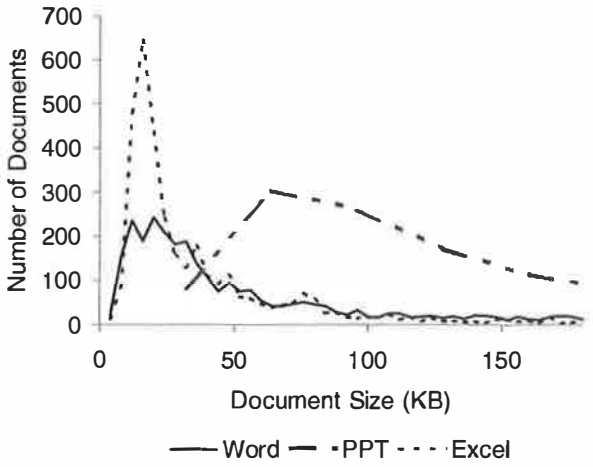


Figure 2: Size distribution of Word, PowerPoint, and Excel documents. Shown are documents with sizes up to 180 KB.

cations other than Microsoft Word. We also eliminated duplicates, removing approximately 5% of our data set.

We converted all the data to Office 2000 formats and we obtained the XML-based representation using Office's OLE Automation interfaces [16]. We wrote a simple Java application that uses OLE Automation to remotely control Office applications to perform data conversions.

Table 1 shows a summary of the documents. For each application, it presents the number of documents, and the number of Web sites from which they originated.

4 Experimental results

This section presents statistics we have measured for Office documents and the components within them. Based on these statistics, we identify opportunities for adapting the documents to bandwidth limited clients.

4.1 Document size

Table 2 shows general statistics for Word, PowerPoint, and Excel documents[1]. The most striking

	Application		
Statistic	Word	PowerPoint	Excel
average (KB)	196.24	891.48	115.02
stdev (KB)	528.44	2145.35	438.70

Table 2: Document size statistics.

aspects of the data are the large average size of the documents and the large standard deviations of our sample.

Figure 2 shows the size distribution of Word, PowerPoint, and Excel documents. The histogram plots documents with sizes up to 180 KB. We observe that the distributions have the same general shape: a cluster around a common small value with a fairly long tail.

Figure 3 characterizes the distributions' tails by plotting document size frequencies for documents larger than 100 KB on a log-log scale. The linear fit[2] of the transformed data ($y \sim x^{-1.7124}$) with $R^2 = 0.8938$ suggests that the tail of the size distribution closely follows a power-law distribution, which is consistent with the large standard deviations of Table 2. The log-log scale histograms for the individual Word, PowerPoint, and Excel documents are not shown here since they are all similar to the cumulative distribution, with linear fits of $y \sim x^{-1.5254}$, $y \sim x^{-1.332}$, $y \sim x^{-1.7485}$, and $R^2 = 0.8612$, $R^2 = 0.8352$, and $R^2 = 0.8226$, respectively.

Interestingly, these results are similar to the findings of Cunha *et. al.* [7] where the size of HTML-based Web documents was found to follow the power-law distribution. However, while Cunha *et. al.* found that most HTML documents are quite small (usually between 256 and 512 bytes), Office documents tend to be much larger. Common sizes of Word and Excel documents size range from 12 KB to 24 KB, and common PowerPoint documents range from 48 KB to 80 KB.

4.2 Size breakdown

Figure 4 shows the breakdown of document sizes for Word documents. For every size category it shows

[1] The document size measurements for table 2 and figures 2 and 3 are based on the *raw* documents retrieved from the Web rather than the normalized Office 2000 translations described in Section 3.

[2] R^2 value ranges from 0 to 1 and reveals how closely the estimated trendline approximates the actual data. The closer the value is to 1, the better the estimate.

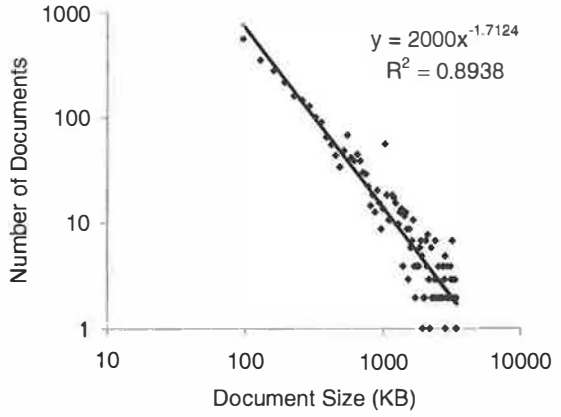


Figure 3: Size distribution of larger Office documents on a log-log scale. Document size frequencies are measured with 16384 byte bins.

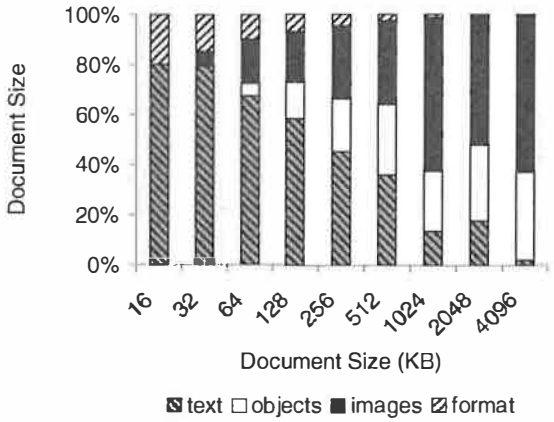


Figure 4: Size breakdown of Word documents. The plot shows that as documents get bigger, images and embedded component data account for most of the document's size.

the contributions of text, formatting information, embedded objects, and images to the documents size. We measured similar breakdowns for PowerPoint and Excel document, but because of space concerns we do not include them in this paper. The PowerPoint documents showed a similar trend to that of figure 4, while in Excel documents the text component accounts for over 95% of the document size in all the size categories.

Figure 4 show that small Word documents are dominated by text and formatting information. For larger Word documents, however, image and embedded component data become the prevalent contributors to document size. This data strongly suggests that efforts to improve access to compound documents should focus on the image and the embedded component data.

One possible optimization would be to remove the embedded component native data from documents that are fetched exclusively for reading. As described in section 2.2, this data is only necessary when editing an embedded component. Users are still able to display the document using the cached image of the component. We measured the savings of this schema and found that it would lead to a reduction in bandwidth requirements for Word and PowerPoint documents as high as 35% and 21%, respectively. PowerPoint documents show less potential benefit because PowerPoint compresses its components data before storing it in the OLE archive, whereas Word does not use compression.

4.3 Comparing OLE archives and XML

The results of our comparison are shown in table 3 and figures 5, 6, and 7. The data reveals that the XML representation can be significantly larger, requiring up to five times more space. XML efficiency is particularly low for small files, which according to our data are the most prevalent. However, XML efficiency improves dramatically as documents get larger.

To understand this, we must consider what happens when a document is converted from an OLE archive to XML. Text and formatting represented in XML takes more space than in Microsoft's internal representation. This explains the inefficiency of XML for small files. However, the XML conversion compresses embedded component data. PowerPoint already compresses its embedded component data, but Word and Excel do not. Because larger documents tend to be mostly images and components (see Figure 4), the XML representation becomes more efficient for large documents, and is even more efficient than the OLE archive for Word documents larger than 1 MB. Excel documents are primarily text and are most efficiently represented as OLE archives.

4.4 Compression

For the OLE archives, we compressed the document by applying gzip to the OLE archive. For the XML

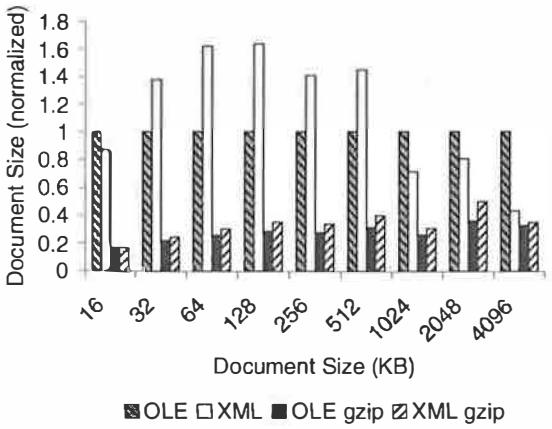


Figure 5: Size distribution of Word documents, with and without compression, for OLE archive and XML formats. Sizes are normalized by the size of the uncompressed OLE archive.

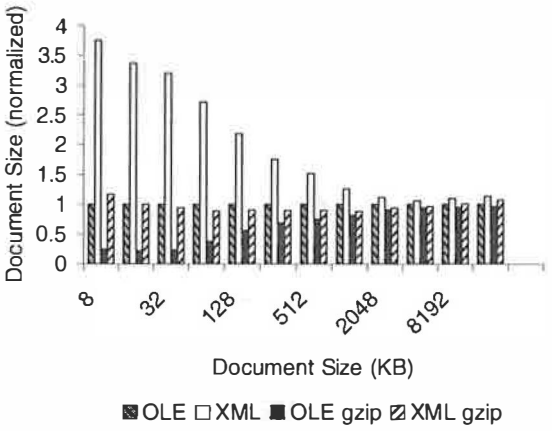


Figure 6: Size distribution of PowerPoint documents, with and without compression, for OLE archive and XML formats. Sizes are normalized by the size of the uncompressed OLE archive.

format, which uses several files, we compressed each file separately. This strategy emulates the potential benefits of a network infrastructure with built-in compression.

The results of these experiments are shown in table 3 and figures 5, 6, and 7. Compression has a dramatic effect on reducing the size of both OLE archives and XML files, achieving savings as high as 77% for the OLE and 90% for XML. Moreover, the difference in size between compressed OLE and compressed XML representations is small enough to be insignificant. This implies that neither representation has an inherent bandwidth advantage when used across a network.

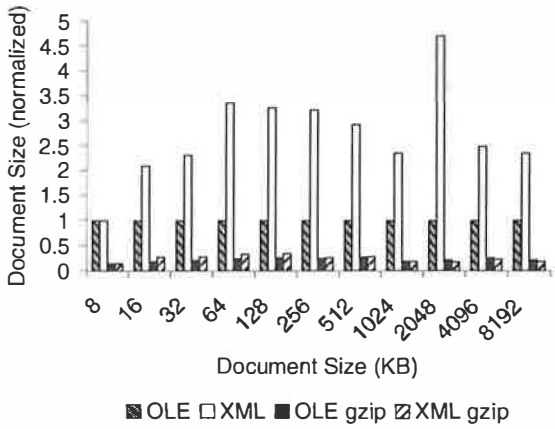


Figure 7: Size distribution of Excel documents, with and without compression, for OLE archive and XML formats. Sizes are normalized by the size of the uncompressed OLE archive.

4.5 Garbage collection

For OLE archives, Office optimizes "save" operations by appending modifications to the end of the file rather than rewriting the whole file every time. While this optimization allows for much faster document saves, it can lead to a significant increase in file sizes. If the user deletes or rewrites a substantial portion of a document and saves it, the original data, now garbage, will be retained. The extra data does not pose a problem for clients accessing the document over random access file systems, enabling the application to skip the dead data. Clients accessing documents over protocols that do not support random access, such as HTTP, are forced to download the whole document before opening it. The end result is fetching extra data that is never used.

In contrast, when a user asks Office to "save as," a new document is written from scratch, without any garbage that may have been in the original document.

We measured the changes in file size for OLE archives by using the "save as" operation. In this experiment we only considered documents that were already in Office 2000 file formats. Other documents are not included because the "save as" operation not only results in garbage collection but also reformats the documents to the Office 2000 formats, which may change document size.

Format	Statistic	Application: Word raw	Word gzip	PowerPoint raw	PowerPoint gzip	Excel raw	Excel gzip
OLE	average (KB)	209.19	61.43	579.53	481.18	110.23	25.67
	stdev (KB)	534.59	248.89	1671.36	1597.20	401.83	97.88
XML	average (KB)	226.43	74.14	795.17	549.03	336.90	28.37
	stdev (KB)	583.79	297.21	1851.92	1713.56	1562.04	92.02

Table 3: Size statistics for documents in raw OLE, OLE compressed with gzip, raw XML, and XML compressed with gzip. The statistics for OLE differ from those presented in Table 2 due to the conversion to Office 2000 formats.

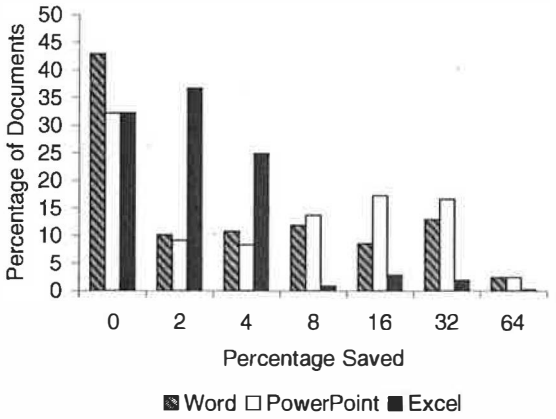


Figure 8: Percentage saved by garbage collection of OLE archive documents.

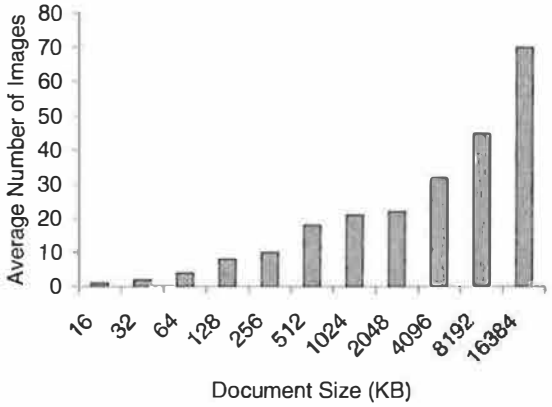


Figure 9: Average number of images in PowerPoint documents.

Statistic	Application: Word	PowerPoint
% of documents with images	34.62	77.01
avg. distinct images	6.01	10.62
avg. image size (KB)	21.58	47.82

Table 4: Images statistics for Word and PowerPoint documents. The table shows the percentage of documents that have at least one images, the average number of images in documents with images, and the average image size.

Figure 8 shows the results of this experiment. Most documents get some benefit from garbage collection. Interestingly, 24% of Word documents and 35% of PowerPoint documents achieve saving greater than 16%.

4.6 Components

In this section we first explore the effects of components on document size. We then present detailed statistics for the three types of components found in Office documents: images, embedded components, and virtual components.

4.6.1 Components and document size

We compared the sizes of Office documents with and without embedded components. Unsurprisingly, documents with embedded components are significantly larger. For example, the average size of Word documents with components is 557.28 KB, relative to an average of 112.32 KB for documents without components. PowerPoint and Excel documents show similar trends: PowerPoint documents average 1334.43 KB with components and 493.58 KB without, and Excel documents average 509.71 KB with components and 109.18 KB without.

4.6.2 Images

Images are the most common type of non-text data found in Office documents. As table 4 shows, 34.62% of Word and 77.01% of PowerPoint documents have at least one image. We do not present

Statistic	Application		
	Word	PowerPoint	Excel
% with components	18.19	46.38	1.42
number of component types	55	11	8
average number of components	6.71	9.18	9.05
average component size (KB)	37.62	18.51	26.01
stdev (KB)	141.78	109.33	133.37

Table 5: Embedded components statistics. The table shows the percentage of documents that have at least one embedded components, the number of different component types, the average number of components in a document, and the average and standard deviation of the size of embedded components.

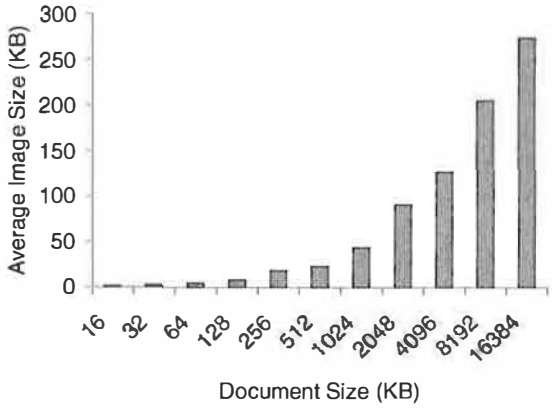


Figure 10: Average image size in PowerPoint documents.

results for Excel documents as very few of them have any images at all.

Figures 9 and 10 show the average number of distinct images and the average size of images for PowerPoint documents. We plot the number of distinct images instead of the total number of images because Office applications cache a single copy of every image regardless of the actual number of times the image appears in the document.

Both plots show similar trends, with increases in the number and size of images as documents get bigger. These results are consistent with the findings of section 4.2, where the size contribution of images to document size becomes the dominant factor as document size increases. The results for Word are similar, and are omitted for brevity.

We compared the average size of images in Office documents to the findings of previous Web studies [2, 23]. In general, these studies report the average size of images between 5 KB and 22 KB. In comparison, Office documents, especially PowerPoint documents, tend to have larger images. These results suggest that image distillation and other adaptation techniques are at least as important for compound documents as they are currently for Web documents.

We measured the reuse of images across our PowerPoint documents by calculating the Adler-32 checksum [8] of the image's data and counting the number of documents that have images with the same signatures. We found that of the 16,189 images embedded in PowerPoint documents, only 14,016 are distinct, while 1,241 images, or 8.85%, appeared in more than one document. We calculated the potential bandwidth savings of a perfect cache for a PowerPoint client reading all the documents in our dataset that came from the same Web site. We found that 26% of the Web sites get some bandwidth savings from the perfect cache, while 11% of the sites see reductions in required bandwidth that are greater than 20 %.

4.6.3 Embedded components

The data in table 5 shows that Office documents are rich in component data, with 18.19% of Word documents and 46.38% of PowerPoint documents having at least one embedded component. Furthermore, the data shows a high diversity of component types, with Word documents having the highest diversity.

Table 6 shows the popularity and average size of component types for Word, PowerPoint, and Excel documents. For all three applications, image components are either the first or second most popular type. Additionally, the average size of image components is among the largest of all types. This evidence further suggests that efforts toward reducing file size should focus on image types.

We observed that for all three applications, the av-

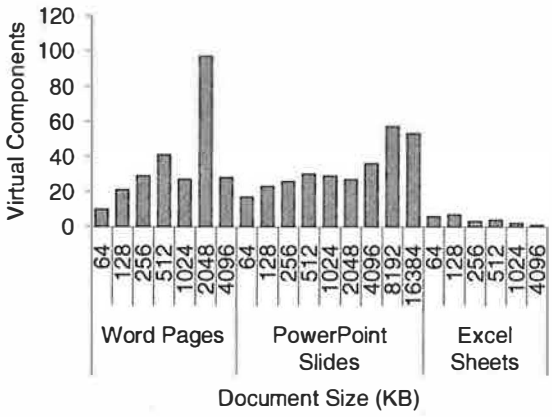


Figure 11: Average number of Word pages, PowerPoint slides, and Excel sheets.

erage number of embedded components and the average size of the components increases as documents get bigger. This trend is similar to the one shown in figure 9 and 10 for images, and is consistent with the findings of section 4.2, where the contribution of embedded components to the document size grows significantly as document size increases.

4.6.4 Virtual components

Table 7 and figure 11 show the average number of pages, slides, and sheets found in Word, PowerPoint and Excel documents. The substantial number of virtual components suggest that Office applications should be adapted to fetch virtual components on demand. To some extent this is already done by the applications when reading OLE archives in random access file systems or by Web browsers reading the XML representation. However, when Office applications open documents where random access is not available or when reading from the XML representation, they download the full document before returning control to the user. While providing universal random access support is likely to prove difficult, we believe that the current Office XML filters can be improved to support on-demand fetching. Alternatively, Puppeteer could provide this type of adaptation. As describe in section 2.1, Puppeteer could fetch the virtual components on demand and use OLE Automation to append them to the application.

Statistic	Word Pages	PowerPoint Slides	Excel Sheets
average	11.95	20.59	5.22
stdev	27.76	17.48	6.49

Table 7: Virtual components. The table shows statistics for pages in Word, slides in PowerPoint, and sheets in Excel documents.

5 Generality of results

In this section we discuss the applicability of our findings to productivity suites other than Office. For this discussion we assume that modern office productivity suites support roughly the same features (embedding, images, etc.) and that document design is driven largely by user needs.

While the average size of documents of different productivity suites is likely to be dependent on the specifics of the applications, the data suggest that the shape of the size distribution of documents would be similar for other productivity suites. We base this claim on the similarities we observed in the size distribution of Word, PowerPoint, and Excel documents. Although each application has a different file format (both for OLE archives and XML), the size of their documents follow the power-law distribution closely.

The breakdowns of document sizes are likely to be similar among productivity suites. While the specifics of how data is divided between images and components might change (*e.g.*, a productivity suite might implement all images as components), the increasing contribution of images and components to document size as documents grow is likely to hold true. Likewise, document design being driven by user needs, the structure of the documents (*i.e.*, the number of pages, slides, etc), the number of embedded components found in documents, and the popularity of images as the most common non-text type of content are likely to be similar.

6 Conclusions and discussion

We characterized compound documents generated by the three most popular applications of the Microsoft Office suite: Word, PowerPoint, and Excel. Our focus was on identifying opportunities

Component	Application: Word Avg. Size (KB)	Word % of Occur.	PowerPoint Avg. Size (KB)	PowerPoint % of Occur.	Excel Avg. Size (KB)	Excel % of Occur.
Equation	0.74	51.12	0.81	1.82		
Other Image			28.68	38.80	4.99	5.00
Word Picture	80.68	14.81	1.31	0.15	2066.83	1.00
Clip Art	10.38	8.93	5.00	41.12	2.56	15.00
Excel Sheet	153.46	8.53	53.18	5.27		
OLE Link	23.80	3.90				
Paint Brush	315.75	1.71			17.91	42.00
MS Draw	7.28	1.35				
PowerPoint	41.74	0.96				
Word			23.28	8.01	28.17	32.00
Graph			3.26	3.86		
Sound			3.35	0.02		
Other	97.52	8.68	8.72	0.94	2.39	5.00

Table 6: Average size and popularity of component types in Word, PowerPoint, and Excel documents.

for adapting these documents to the constraints of bandwidth-limited clients. Our study encompassed over 12,500 documents, comprising over 4 GB of data, retrieved from 935 different Web sites.

We identified the following opportunities for adaptation:

1. For large documents, images and components account for the majority of the data. Moreover, images and image components are the most common non-text data found in Office documents. These results suggest that components, and in particular images should be the main focus of any adaptation efforts. We are currently in the process of adding quality-aware transcoding and caching of images and components to Puppeteer and plan to measure the savings of these techniques.

2. For read only documents, discarding the native component data results in savings of up to 35% and 21% for Word and PowerPoint respectively.

3. Garbage collection of OLE archives achieves savings greater than 16% for 24% of Word and 35% of PowerPoint documents.

4. Compression achieves savings of 77% for OLE archives and 90% for XML. Moreover, once compressed there is no significant difference in the sizes of the two file formats. Since XML formats are significantly easier to parse and manipulate than OLE archives, they are a more attractive target for adaptation.

5. The structure of Office documents (pages, slides, and sheets) can be used to download elements on demand and reduce the time that users wait before they can start work on the document.

Furthermore, our experience studying the Office file formats resulted in the following insights:

1. The data suggests that the "save as" operation is largely misunderstood by users. The large savings that we show from garbage collection suggest that users do not understand the implications of *fast-save* mode (the default), instead believing the "save as" operation to be a way to create a copy of the document.

2. The lack of built-in support for compression in OLE archives has forced designers to implement ad-hoc solutions to achieve high performance. This experience suggests that a compression feature would be a desirable addition to OLE archives.

3. OLE archive formats are likely to remain the preferred intermediate format for Office documents, while the XML-based format will likely be the format of choice for Web publishing. The XML-based format has the advantage that it can more easily be interpreted by application other than Office (*e.g.*, Web browsers). It is also amenable to widespread browser techniques that improve user perceived latency, such as incremental rendering and fetch on-demand. On the flip side, the current imple-

mentation of Office 2000 does not implement incremental loading or writing of XML-based documents, leading to higher latencies for opening and storing XML-based documents than those experienced on similar OLE archive documents. Moreover, some of the Office formats do not yet have XML equivalents.

References

[1] Alta Vista home page. http://www.altavista.com.

[2] ARLITT, M. F., AND WILLIAMSON, C. L. Server workload characterization: the search for invariants. In *ACM SIGMETRICS Conference* (Philadelphia, Pennsylvania, 1996).

[3] BAGRODIA, R., CHU, W. W., KLEINROCK, L., AND POPEK, G. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications 2*, 6 (Dec. 1995), 14–27.

[4] BRAY, T. Measuring the Web. In *The World Wide Web Journal* (1996), vol. 1-3.

[5] BROCKSCHMIDT, K. *Inside OLE.* Microsoft Press, 1995.

[6] CHAPPELL, D. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[7] CUNHA, C. R., BESTAVROS, A., AND CROVELLA, M. E. Characteristics of WWW client-based traces. Tech. Rep. TR-95-010, Boston University, Apr. 1995.

[8] DEUTSCH, P., AND GAILLY, J. L. ZLIB compressed data format specification version 3.3. `http://src.doc.ic.ac.uk/Mirrors/ftp.cdrom.com/pub/infozip/doc/rfc1950.txt`, May 1996.

[9] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., AND BREWER, E. A. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications 5*, 4 (Aug. 1998), 10–19.

[10] JOSEPH, A. D., DELESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., AND KAASHOEK, M. F. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain Resort, Colorado, Dec. 1995), pp. 156–171.

[11] KATZ, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications 1*, 1 (1994), 6–17.

[12] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[13] MICROSOFT CORPORATION. *Microsoft Excel File Format.* Redmond, Washington, 1997. MSDN Online, `http://msdn.microsoft.com`.

[14] MICROSOFT CORPORATION. *Microsoft PowerPoint File Format.* Redmond, Washington, 1997. MSDN Online, `http://msdn.microsoft.com`.

[15] MICROSOFT CORPORATION. *Microsoft Word File Format.* Redmond, Washington, 1997. MSDN Online, `http://msdn.microsoft.com`.

[16] MICROSOFT PRESS. *Microsoft Office 97 / Visual Basic Programmer's Guide*, 1997.

[17] MICROSOFT PRESS. *Microsoft Office 2000 / Visual Basic Programmer's Guide*, 1999.

[18] MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, Colorado, Dec. 1995).

[19] NIKSIC, H. Gnu Wget. `http://www.gnu.org/manual/wget/ps/wget.ps`, Sept. 1998.

[20] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM) 51*, 5 (Dec. 1997), 276–287.

[21] PITKOW, J. E. Summary of WWW characterizations. In *Proceedings of the Seventh International World Wide Web Conference* (Brisbane, Australia, Apr. 1998).

[22] SATYANARAYANAN, M. Hot topics: Mobile computing. *IEEE Computer 26*, 9 (Sept. 1993), 81–82.

[23] SEDAYAO, J. "Mosaic will kill my network!" - studying network traffic patterns of Mosaic use. In *Proc. of the 2nd International WWW Conference* (Chicago, Illinois, 1994).

[24] WOODRUFF, A., AOKI, P. M., BREWER, E., GAUTHIER, P., AND ROWE, L. A. An investigation of documents from the World Wide Web. In *The World Wide Web Journal* (1996), vol. 1-3.

A Toolkit for Building Dependable and Extensible Home Networking Applications

Yi-Min Wang
Microsoft Research
Redmond, WA

Wilf Russell
Microsoft Research
Redmond, WA

Anish Arora
Ohio State University
Columbus, OH

Abstract

Dependability and extensibility are two of the key requirements to successful home networking. In this paper, we describe the design and implementation of a software development toolkit for building dependable and extensible home networking applications. A reliable Soft-State Store (SSS) is implemented as a shared infrastructure to simplify robust distributed programming against device and object failures. SSS supports multi-timescale refreshes and selectively uses persistence to accommodate the battery power and network bandwidth constraints in the home networking environment. A publish/subscribe event system allows any changes in the SSS to be propagated to interested subscribers, which then perform appropriate adaptive, corrective, alerting, or cleanup actions. An Attribute-Based Lookup Service (ABLS) and a Name-Based Lookup Service (NBLS), both implemented on top of the SSS for robustness, provide a level of indirection for supporting extensibility as well as allowing user-friendly, natural language-based access. We demonstrate the use of the toolkit for building a home networking system in an actual deployment. We describe two end-to-end remote home automation applications, present performance results, and report our experiences.

1. Introduction

The success of the Web has demonstrated the great power of being connected. As the world increasingly moves towards a fully connected one, home networking that connects smart household appliances together and connects them to the Internet becomes the natural next step. The simplest form of home networking has emerged to allow the sharing of files, printers, and Internet connections, and to enable networked PC games. The next wave of advanced home networking applications will include family communications, device automation, digital Audio/Video (A/V) distribution, remote maintenance of household appliances, etc.

In the *Aladdin* home networking project, we focus on building end-to-end user scenarios and using those to drive the design and implementation of the required system infrastructure. Based on our experience in implementing and deploying a home networking system in a house with real users, we have identified dependability, extensibility, user-friendly interface, and remote access capability as the four key requirements to useful and successful home networking. *Dependability* ensures that failures of hardware devices and software objects will be detected, appropriate recovery or cleanup actions will be performed, and homeowners will be alerted if necessary. *Extensibility* allows any new device to be plugged into any of the in-home networks (phoneline, powerline, wireless, etc.) and become available to all existing applications. *User-friendly interface* allows users to control appliances and retrieve information in a natural way. *Remote access capability* lets homeowners connect to their homes at any time, from any place, and on any device.

In this paper, we give an overview of the Aladdin system and software architectures for addressing the above issues. We then focus our attention on the system infrastructure provided by the toolkit that we have constructed to simplify the task of building dependable and extensible home networking applications. Finally, we describe two end-to-end remote home automation applications to demonstrate the power of the toolkit and to evaluate its performance.

Compared to traditional networked environments, the home networking environment is more heterogeneous and dynamic. It is heterogeneous because consumer devices manufactured by different vendors, connected to different networks, and running different protocols are likely to coexist. It is dynamic because these consumer devices tend to connect, disconnect, move, fail, etc. more often. We propose a *Soft-State Store* with *Publish/Subscribe eventing* and lookup services built on top of the store as a uniform framework for robust management of diverse devices, where *soft-state* is defined as *volatile or nonvolatile states that will expire if not refreshed within a pre-determined, but configurable, amount of time.* Both hardware devices and software objects periodically announce their existence and optionally their states to the soft-state store, which may consist of multiple individual stores distributed throughout the system. When a device/object fails or gets disconnected, its corresponding soft-states eventually time out. Such changes to the soft-state store generate events to all interested subscribers, which then perform appropriate actions to adapt to the changes.

The paper is organized as follows. Section 2 gives an overview of the system and software architectures of the Aladdin home networking system. Section 3 describes the design and implementation of the soft-state store, the event system, and the lookup services. Section 4 describes two remote home automation applications built with the toolkit. Section 5 reports our experiences from the actual usage of the system. Section 6 discusses related work, and Section 7 summarizes the paper.

2. Overview of the Aladdin System

2.1. Distributed System Architecture for Home Networking

Figure 1(a) illustrates an ideal home networking system where the house is wired for running Ethernet and most devices are smart, networked devices connected directly to the Ethernet and running device control software themselves. A *home gateway* machine sits between the home network and the external communication infrastructures including the Internet and telephony. *User Access Points (UAPs)* are wall-mounted or stand-alone flat-panel displays deployed throughout the house to allow convenient access to in-home information (calendars, etc.) as well as the Internet from anywhere in the house. UAPs also expose Web-based, natural language-based, and voice-based interfaces for remotely controlling household devices and for monitoring environmental factors through remote sensors. Network bridges are provided for bridging devices on other communication media such as the powerline, Radio Frequency (RF), InfraRed (IR), and A/V cables to the Ethernet backbone. Such devices do not directly connect to the Ethernet for various reasons including legacy, cost, security, market competition, etc.

Since smart devices are not yet generally available, the current Aladdin system accommodates existing devices by using multiple Windows 98 PCs and their peripherals to serve as both User Access Points and network bridges, as shown in Figure 1(b). (Six PCs are used in this deployment.) The PCs also act as device proxies by running device control software on behalf of the devices. The system is deployed in the first author's three-story house and used by the author on a daily basis.

The PCs are all connected by 1Mbps to 10Mbps Ethernet over the phoneline [H98]. Since the powerline in the house has serious signal attenuation problem that prevents the low-cost consumer powerline devices from reliably communicating between any two outlets, powerline control requests are first routed to the PC that is (electrically) closest to the target device and then bridged to the powerline. Similarly, battery-operated RF devices such as the motion sensors and door sensors are usually short-ranged to preserve battery power. RF signals transmitted by these sensors are received by PCs within their ranges and bridged to the phoneline network to reach other parts of the house. Although currently most Aladdin applications do not require the full processing power of the PCs, they will be useful in the future for running potentially computation-intensive smart-home software including location tracking, person identification, voice recognition, learning, etc. [S+98][M98][Es99]. In total, the current Aladdin system consists of about 60 devices.

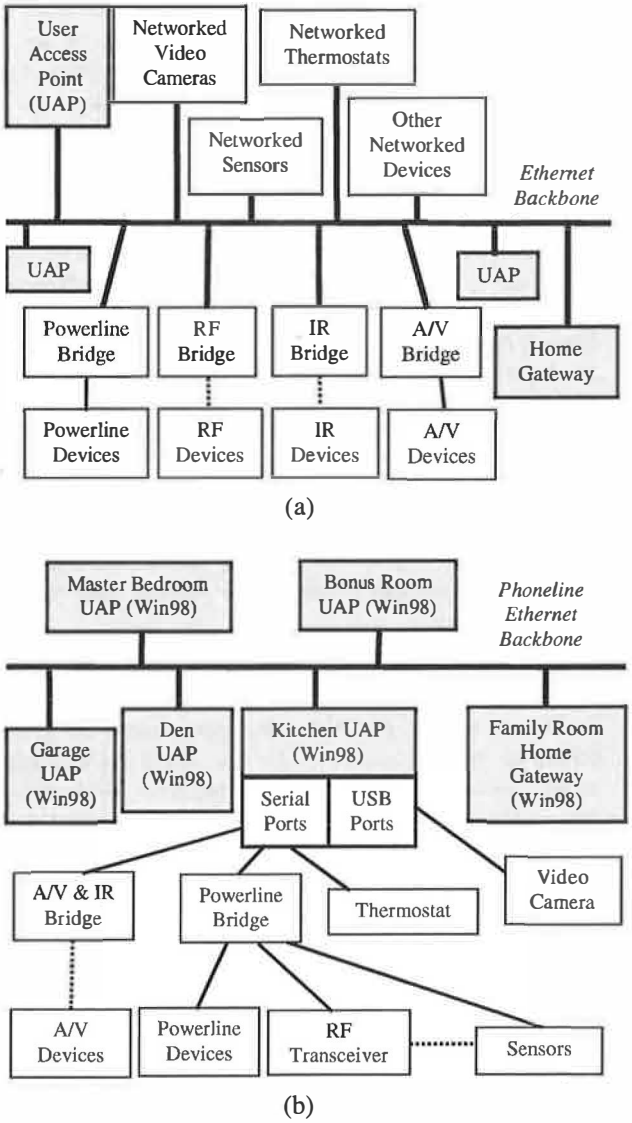


Figure 1. Distributed system architecture of the Aladdin home networking system. (a) Ideal future architecture; (b) Current architecture (only the peripherals of the

kitchen UAP are shown; all the other UAPs have similar setup).

2.2. Software Architecture

Figure 2 shows the overall software architecture of the Aladdin system, which provides an abstraction over the hardware devices, and connects them to external communication infrastructure. We give a brief overview of all three layers in this section, and will focus on the system infrastructure layer in the remainder of the paper.

1. System infrastructure: At the bottom layer, the system infrastructure consists of five components. The soft-state store manages the lifetime and replication of soft-state variables across individual data stores on each machine. It supports a Pub/Sub eventing mechanism that allows programs to subscribe to events related to changes in the store. The attribute-based lookup service maintains a database of all available devices and supports queries based on device attributes such as device type, physical location, etc. The name-based lookup service maintains a table of all running object instances and supports simple name-to-object-address mapping. The device announcement protocol describes how devices not connected to the main Ethernet announce their existence to the attribute-based lookup service. An Aladdin Device Adapter has been built to implement an instance of that protocol for powerline devices. Finally, the system management daemons are responsible for detecting the failures of PCs and devices, and initiating recovery actions. We do not cover this last infrastructure component in this paper.

2. Application layer: There are two types of home networking applications in the current Aladdin system. In the control-type scenario, the applications receive user requests as input, consult the lookup services to identify the devices and device objects that should be involved, and perform actions on them to satisfy the request. Device objects encapsulate device- and network-specific details and present interfaces (sets of method calls) as the programming abstraction for device control. Examples are camera objects for taking snapshots and recording video clips, garage door opener objects for operating the garage doors, etc.

In the sensing-type scenario, the applications monitor a list of environmental factors and take actions when any of the monitored events happens. Through the ABLS, the applications identify the appropriate events to subscribe to. Independently, device daemons act as proxies for sensors by monitoring sensor signals and updating appropriate soft-states to trigger events. For example, an alerting application running on every User Access Point subscribes to all events corresponding to critical sensors (water sensors, temperature sensors, safe-box sensors, etc.) and will sound alerts when any of the sensors fires.

Most commercial home networking products for intelligent sensing and control tend to be closed solutions limited to a single communication medium and are not extensible. Aladdin's approach of relying on the soft-state store, pub/sub eventing, and lookup services to tie together devices connected to different communication media provides a unique opportunity for constructing versatile applications.

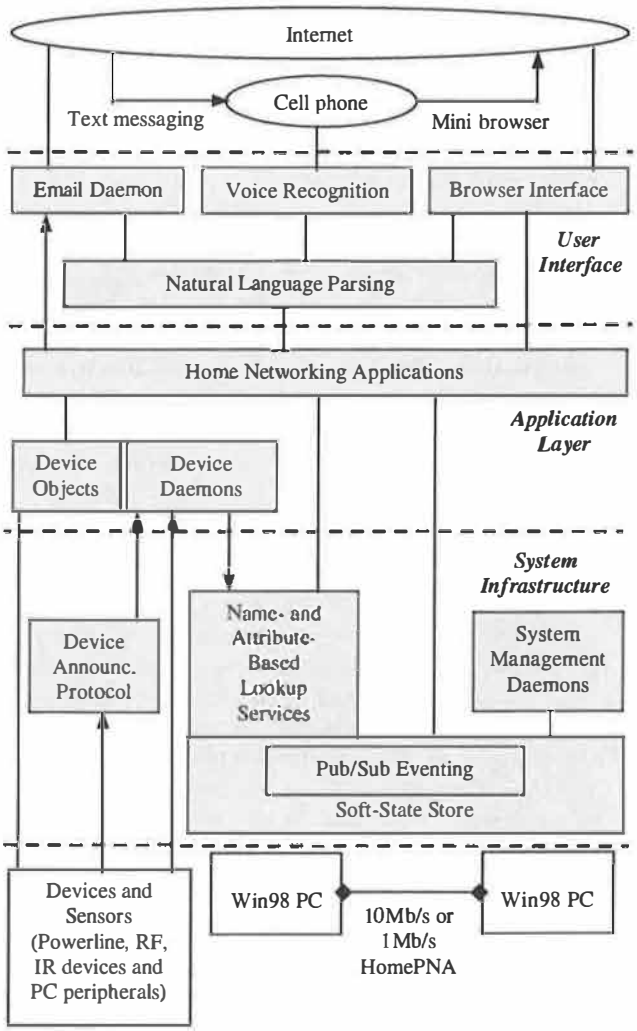


Figure 2. Software architecture of the Aladdin home networking system.

3. User interface: Unlike most other distributed systems, home networking systems are to be used by naïve computer users and so providing friendly user interface is especially important. The current Aladdin system supports three forms of user interface: a *browser interface* that allows the user to browse through all available devices or select devices based on attributes,

and to control devices through point-and-click; a *text-based natural language interface* based on a limited but customizable vocabulary; and a *voice-based interface* that employs speech recognition technology based on the same vocabulary. All three forms are available for in-home use. They are also being extended to support remote home automation when the user is away from home. When DSL or cable modem is available, the same browser interface can be used from remote locations. The text-based natural language interface has been extended to an email-based remote home automation interface. The user can send an email containing a control request to an account hosted by an Internet Service Provider (ISP). (Current deployment uses Microsoft Network, MSN.) The Aladdin email daemon periodically dials up the ISP, retrieves and parses the request, performs the actions, and sends a reply email that may optionally contain video clip(s) confirming the actions. In addition to the standard security mechanisms such as digital signatures and data encryption, the home control vocabulary can be customized to provide additional security. Through the text messaging support provided by cell phones, the email daemon can almost synchronously alert the users wherever they are when, for example, any of the critical sensors fires at home. Work on extending the voice-based interface to work reliably over telephony is still in progress.

3. Home Networking Toolkit

The Aladdin home networking toolkit currently consists of seventeen thousand lines of code packaged as several DLLs and EXEs. In this section, we describe the motivation and design of each component, followed by their specific implementations on Windows 98 and the Application Programming Interfaces (APIs). The soft-state store manages the timeouts and propagation of soft-state variables, and provides a generic publish/subscribe eventing mechanism that reflects any changes to the store. The name-based lookup service is implemented on top of the soft-state store, which actually stores and maintains the table of lookup service entries. In contrast, the attribute-based lookup service stores the entries in a database to support queries, while creating a soft-state variable for each entry to manage the timeouts and eventing. Both lookup services register an event callback conversion module with the soft-state store to convert generic events into their respective domain-specific events. Most applications interact with the system through higher-level lookup services APIs. But the low-level soft-state store APIs are also available.

3.1. Soft-State Store with Eventing

Consider the common requirements for the following scenarios: when a battery-operated garage-door sensor runs out of battery, the system should detect the failure and alert the user; also, it should discard the previous state of the sensor so that home networking applications do not perform erroneous actions based on stale data. When a device suddenly gets disconnected from the system and when an object gets terminated abruptly, their corresponding entries in the lookup services should eventually expire to allow the system to reclaim the space of those entries and to minimize the chance of client applications getting stale information and potentially malfunctioning. When an essential daemon process fails either due to machine crash or process termination, the system should be able to detect the failure and either reset the machine or restart the daemon.

To address the above issues, either the system needs to *ping* the sensor/device/object/daemon or the latter need to send periodic *heartbeats* or *refreshes* to the system. Traditionally, systems with homogeneous and relatively static parts are maintained in terms of *"hard"-states* that are updated upon demand by pinging the parts. By way of contrast, in the heterogeneous and dynamic environment of home networking, it is preferable to maintain systems in terms of *"soft"-states* that are updated periodically by refreshes from the parts.

There are several reasons for preferring soft-states to hard-states in Aladdin. First, to keep the dollar cost low, many consumer sensors are transmitters only and do not support pinging of their status. Second, since many network protocols, object models, and programming paradigms (distributed objects, wire protocols, etc.) are likely to coexist in home networking systems due to market competition, it will not be practical to require the system to ping all devices and objects with various protocols, models and paradigms. Third, the devices/objects being pinged may hang the pinging operations, thereby complicating system robustness. Finally, hard-states complicate the recovery tasks of daemons and protocols upon system crashes. By relying on the refreshes to reconstruct lost data, soft-states greatly simplify the recovery tasks. By the same token, they also simplify maintenance of information about devices that leave the system spontaneously and without announcements.

To simplify the development of home networking applications based on soft-states, we implemented a *Soft-State Store (SSS)* that serves as a shared infrastructure for managing and propagating soft-states across different applications and machines. A provider of a soft-state variable specifies the *refresh interval* and

the *threshold number of missing refreshes* before the variable is timed out. The SSS is responsible for maintaining the time-to-live timers for all providers.

An event system is built on top of the SSS to allow applications to subscribe to events related to data changes in the SSS. Subscribers of events can be notified, for example, of (1) the failure of a critical system daemon so as to perform fail-over recovery, (2) the addition of a new device so as to adapt to the new overall device availability, (3) the failure of a device so as to raise an alert to the users, and (4) the disappearance of a resource-consuming entity so as to perform cleanup actions. Subscriptions themselves are maintained in the SSS and periodically refreshed by the subscribers.

As will be demonstrated in the lookup service section, some of the refreshes in the home networking environment cannot be performed with high frequencies. Low-frequency refreshes ranging from a couple of hours to a day may be necessary to accommodate power constraints of battery-operated commodity devices and sensors, and the bandwidth limitations of some of the in-home networks. This is in contrast to the conventional case where the refresh rate is as high as every few seconds or minutes, as a result of which it is adequate to store the soft-states in volatile memory and, upon a system failure, to rely completely on the high-frequency refreshes for recovery.

To ensure acceptable data quality for soft-states whose refresh frequency is low, we introduce the notion of *persistent soft-states*. By maintaining these soft-states in persistent storage, the SSS can recover them after total system failures without waiting for a long time until the next refresh. For example, reliable remote operation of the garage door can be resumed immediately after system failure without waiting for the door sensors to refresh their states in up to 90 minutes. One must, however, pay special attention not to restore potentially stale, persisted soft-states. Each persisted soft-state variable is time-stamped with the *latest refresh time*, and *the number of missing refreshes* is also recorded. When the persistent file is used after a recovery to restore part of the soft-state store, the current time is compared with the time stamp of each variable and the time difference divided by the refresh interval is used to increase the number of missing refreshes. That is, any potential refreshes that may have happened during the system downtime are treated as missing refreshes. Any variable with the adjusted number of missing refreshes exceeding the threshold is not restored.

The Aladdin system is subject to a variety of faults that are weaker than total system failures but must be considered as they affect the freshness and availability of SSS data. At least one PC, typically the home gateway machine, is connected to an Uninterruptible Power Supply (UPS) so that short power outages do not cause total system failures. Upon a PC reboot, SSS maintains a leader that allows the rebooted PC to catch up with the SSS replicas: the leader streams its cache of soft-states to the rebooted PC over the phoneline Ethernet, thus minimizing the staleness of SSS data and the outage period of SSS service at the latter.

Implementation

The soft-state store implementation consists of a COM (Component Object Model [B98]) EXE server and a client-side COM DLL. It supports two interfaces. The ***ISoftStateStoreAdmin*** interface contains the following methods:

- *RegisterSoftStateTypes()* and *RemoveSoftStateTypes()* for defining and removing custom types and sub-types of soft-state variables, respectively; for example, the CRITICAL_SENSOR type is registered as a subtype of the SENSOR type.
- *ChangeTimeoutOnVar()* and *ChangeTimeoutOnSubscription()* for manipulating the metadata of soft-state variables and event subscriptions. Specifically, they allow changes to be made to the refresh intervals and the threshold number of missing refreshes.

The second interface, ***ISoftStateStore***, consists of:

- *RegisterSoftStateVars()* and *RemoveSoftStateVars()* for creating and deleting individual soft-state variables of particular types, respectively. *RegisterSoftStateVars()* also takes as input a flag indicating whether auto-refreshes are requested. If the flag is set to true, the client-side SSS DLL automatically sends periodic refreshes on the client's behalf. When the client application terminates without removing its variables, the DLL also dies and the variables will eventually time out.
- *SetValue()* and *GetValue()* for setting and retrieving the values of variables, respectively; *GetValue()* can optionally return the current number of missing refreshes and time to next planned refresh. Applications can use such information on potential data staleness to perform different actions, if desirable.
- *GetVarsOfType()* returns the names and values of all soft-state variables and optionally subtypes of a particular type.
- *SubscribeEvents()* and *UnsubscribeEvents()* for subscribing and unsubscribing events related to the changes of individual soft-state variables or all variables of certain types. The subscriptions

themselves are soft-states as well so that the system can remove stale subscriptions when the subscribers terminate without performing proper cleanup. The subscriptions, however, are only local and not replicated to other machines.

The callback address supplied as part of the input parameters to a *SubscribeEvents()* call is in the form of a unique NBLS name, to be described shortly, to allow late binding. When the SSS needs to fire an event to a subscriber, it resolves the name at that time through the NBLS to obtain the subscriber's up-to-date addresses. This late-binding model is essential for coping with failures, recoveries, and mobility of the subscribers, and is particularly important for subscriptions to rare events. If a subscriber uses the COM programming paradigm, it must support the *ISSSNotify* interface and announce the (marshaled) interface pointer to NBLS. ***ISSSNotify*** consists of four methods:

- *Added()* is invoked when a new soft-state variable is added to a subscribed type;
- *Changed()* and *Deleted()* are invoked when a variable or type is updated and deleted, respectively. A flag in the *Deleted()* call indicates if the deletion was the result of an explicit removal operation or expiration due to missing refreshes.
- *MetaUpdate()* notifies the subscriber that metadata associated with the variables or types have been changed. This is especially useful should the system need to change the timeout parameters when demand for a particular soft-state variable decreases, SSS load increases, or A/V applications demanding high bandwidth are being started.

3.2. Lookup Services

Lookup services are the key to extensibility. By providing one level of indirection, lookup services allow devices and objects to dynamically join the system and be available to client applications. In the home networking environment, we have found it useful to divide the lookup services into two layers. At the upper layer, the *Attribute-Based Lookup Service (ABLS)* maintains a database of available devices, sensors, installed software modules, etc. It supports queries based on a combination of attributes and returns a list of unique names (called *NBLS names*) identifying the matching items. For example, the query "device=curtain and floor=1" returns the names of all first-floor curtains. At the lower layer, the *Name-Based Lookup Service (NBLS)* maintains a table of available software objects. It takes a unique name as input and returns a list of "addresses" that can be used to contact the target object. For example, the unique name "living_room_curtain" may generate an NBLS response that contains two addresses for reaching the curtain object: one can be unmarshaled into a DCOM pointer for making synchronous calls, and another one can be unmarshaled into a handle to a message queue for making asynchronous calls. Both ABLS and NBLS rely on the soft-state store for robustness against sudden disappearances of devices and objects.

3.2.1. Attribute-Based Lookup Service (ABLS)

Attribute-based lookup service provides the foundation for user-friendly naming of devices. Instead of identifying each device by its low-level communication address, an Aladdin user identifies each device by its *physical location* in the house, which is a notion most familiar to the user and especially useful for remote automation; for example, "the lamp on the garage side of the kitchen" or "the VCR on the second floor". Since most of the devices in the current Aladdin system are ordinary, non-Ethernet devices, the main challenge is to devise a mechanism that would allow such devices to announce their attributes into the ABLS and to automatically include their physical location information in the announcement.

To address the above issue, we introduce the concept of an *Aladdin Device Adapter (ADA)*, which serves as the representative of an attaching device to participate in the Aladdin lookup service-based system. To make the presentation more concrete, we will describe the ADA in the specific context of the X10 powerline control protocol [S98]. The general concept is applicable to other communication media and protocols as well.

The X10 protocol allows for 256 unique addresses, each of which is specified by a *house code* (A through P) and a *unit code* (1 through 16). To allow a device to announce its physical location, we perform a one-time configuration task by *assigning a unique X10 address to every outlet that the user would like to control*, and *storing in ABLS the mapping of the address to the set of physical location attributes associated with the outlet*. For example, the address K3 may be assigned to an outlet on the garage side of the kitchen. To connect an ordinary device to an outlet, the device is plugged into an Aladdin Device Adapter and the Adapter is plugged into the outlet. Similar to the common X10 receiver modules, the Adapter has two dials for setting its X10 address, which in our scheme must be set to the address assigned to the outlet. The Adapter also has a third dial for selecting a *device code* among a pre-defined set of codes ("1" for lamps, "2" for fans, etc.) to indicate the type of the attaching device.

The Aladdin Device Adapter performs three tasks: *announce*, *revoke*, and *refresh*. When the device is switched on (and hence available for remote control),

the Adapter detects that through an AC current sensor and announces the device code and the X10 address over the powerline in the form of an extended X10 code. Device daemons running on a subset of PCs receive such an announcement and register with the ABLS (see Figure 2) the received device code and X10 address, which gets translated into the device's physical location information. The PCs also register themselves as the proxy controllers that can be contacted to instantiate appropriate device objects to control the device. When the device is broken or unplugged from the Adapter, the Adapter detects that through the current sensor and sends out another extended X10 code to indicate that the device has left the system. The receiving PCs then contact the ABLS to remove the device entry. The Adapter is also responsible for sending periodic announcements so that the proxy controllers can refresh the soft-state ABLS entry for the device. When the Adapter itself is unplugged from the outlet, the periodic refreshes stop and the device's ABLS entry will eventually be timed out to reflect the fact. Since powerline has limited bandwidth and devices do not join and leave very often, the refresh rates used by the Adapters are typically in the order of hours.

Sensors refresh their ABLS entries through a similar bridging process performed by the device daemon. Some existing low-cost consumer sensors already follow the soft-state model by sending periodic announcements approximately every 90 minutes. To participate in the ADA protocol, they need to be enhanced with a third dial for selecting the sensor type code. Since many consumer sensors operate on batteries to allow flexible installation, such low-frequency soft-state refreshes are particularly important for conserving battery power. Sensor refreshes in fact announce more than just the existence of a sensor; they also provide the current state of the sensor since most consumer sensors do not support polling of their states. Such state information also resides in ABLS and will expire at the same time the associated lookup service entry expires to prevent any client applications from accessing the stale state of a broken sensor.

Implementation

We took the wire protocol approach in the design of ABLS. Any smart device that plugs into the phoneline Ethernet can interact with the ABLS directly by generating messages conforming to the wire protocol. A COM API layer is defined on top of the wire protocol to simplify ABLS programming on PCs, currently the only Ethernet devices in the system.

The ABLS wire protocol specifies the XML tags for the request and response of two groups of operations: administrative operations and service operations. The first group includes operations involving the definition of the vocabulary for physical location attributes, the mapping of location names to sets of location attributes, the assignment of device type codes, the registration of ABLS data types for event subscriptions, etc. The second group consists of operations for announcing/refreshing and removing device entries, for submitting device queries in the form of attribute-value pairs, and for subscribing and unsubscribing events related to changes in ABLS.

ABLS requests are multicast to all ABLS daemon replicas running on a subset of PCs and listening on a well-known multicast address and port. All active daemons perform updates if necessary, but only the leader sends a response. The current Aladdin implementation uses the Jet database engine [HF95] to provide persistent storage of ABLS entries and to support SQL-like queries. The soft-state timers for the entries are maintained separately by the SSS.

To simplify ABLS programming, a client-side DLL provides a COM API layer that takes application-specific data as input and generates ABLS requests according to the wire protocol. The APIs are factored into two interfaces: ***IABLSAdmin*** and ***IABLSServices***. For example, *IABLSAdmin::SetLocaleInfo()* is typically used to specify the mapping between an X10 address and a set of physical location attributes; *IABLSAdmin::SetDeviceInfo()* assigns a device code to a particular device type; *IABLSServices::Update()* is used to indicate that a device of a particular device type is now controllable at a communication address. It also specifies the host name of the controlling PC.

IABLSServices::LookUp() takes as input a list of attribute-value pairs and returns the number of matching device entries and an *Enum* structure that can be enumerated to retrieve all matching entries. Each entry may contain multiple subentries, each of which is associated with a PC candidate that can be contacted to control the device and is identified by a unique *NBLS name*. For each matching device, the client application then chooses one of the NBLS names and submits it to the NBLS to locate the corresponding device object. To simplify the programming for the common case, an entry from the *Enum* supports a *GetAddresses()* method that implicitly performs the NBLS lookup using the NBLS name of the first subentry. In addition, *GetAddresses()* takes as input a flag indicating whether automatic activation is requested. If the flag is set and the object instance is not running, the ABLS will, based on the information supplied by *IABLSServices::Update()*, create the object instance on an appropriate machine and return the list of addresses to the caller.

3.2.2. Name-Based Lookup Service (NBLS)

The name-based lookup service essentially provides a lookup table that maps each unique name to the object instance identified by that name. The Aladdin NBLS, however, has several unique features that differentiate it from other name services and object locator services. First, it is built on top of the soft-state store and therefore robust against object failures and non-graceful termination. Client applications can take advantage of the publish/subscribe eventing mechanism to request to be notified when a target object is instantiated and announced to the NBLS, without having to periodically poll the service. Object instances that, for example, run on battery-operated devices may wish to perform refreshes at a low rate. Due to the persistence support of the SSS, such objects can remain available to clients through the lookup service immediately after the NBLS fails and recovers.

The second feature of the NBLS is its extensibility. Most device objects in the current Aladdin system have been built as DCOM objects because the object-oriented RPC model greatly simplified programming. However, several limitations of DCOM have been identified. For example, the synchronous RPC model does not work well in the presence of failures and in loosely coupled environments; the requirement of proxy/stub installation creates complexity, hinders application evolution, and creates versioning problems; the binary marshalling format and the private wire protocol do not work well in an open environment and make it hard to do introspection. To allow the Aladdin system to evolve as the distributed computing world moves towards loosely-coupled, asynchronous messaging, we built extensibility into NBLS, instead of making it a mere object locator service for DCOM. The heterogeneous nature of the home networking environment further calls for the need of an extensible lookup service. Future generations of smart devices may plug into different communication media and run different communication protocols. Even for devices based on the same communication protocol, market competitions from different vendors may eventually fragment the market and require devices running software with different object models or programming paradigms (distributed objects, wire protocols, etc.) to coexist. To maximize market acceptance, some devices may choose to support multiple protocols, models, and paradigms.

To provide extensibility, the Aladdin NBLS allows mapping a unique name to potentially multiple addresses. Each address starts with a prefix, identifying the protocol/model/paradigm, followed by an opaque string that encapsulates addressing information. Upon receiving a list of addresses as the response to an NBLS query, the client application can enumerate the list to identify the addresses that it can understand, choose the one that is most desirable, and unmarshal the address into a communication handle if necessary.

As an example, a DCOM object may choose to support an additional address for queued, asynchronous communication in order to accommodate non-RPC clients. The refresh interval for the synchronous address (i.e., a marshaled DCOM interface pointer) is typically in the range of tens of seconds to allow fast detection of failed objects. The interval for the queued address is usually longer (minutes to tens of minutes) to accommodate failures and recoveries. When the object's hosting machine fails and reboots, the object is destroyed and its synchronous address soon gets timed out. If a client wants to send a request to the object at that time and a queued call is acceptable, it can do so by using the remaining queued address. When the object restarts, it will check its message queue and process the request, much like a person walks into his/her office in the morning and checks voice mail or email.

Implementation

The NBLS wire protocol uses similar semantics to an earlier version of the Simple Service Discovery Protocol (SSDP) [G+99]. It specifies the XML tags for the request and response of device/object announcements, revocations, and lookups. Similar to the ABLS, a client-side DLL exposes a COM interface ***INBLSServices*** to simplify NBLS programming on PCs. For DCOM applications, a second client-side DLL provides additional helper APIs to further simplify programming: *CreateDCOMRefDisplayName()* marshals a DCOM interface pointer into an address string with the "*DCOMRef*" prefix; *BindToDisplayName()* takes such an address string as input and unmarshals it back to an interface pointer. On the server side, NBLS is tightly coupled with the SSS and relies on the SSS to store the lookup service entries as well as to manage timeouts.

4. Applications and Performance Results

We now describe two of the home networking applications that we have built using the toolkit: email-based remote automation and cell phone-based remote notification. We demonstrate how the various system infrastructure components are involved in these end-to-end application scenarios and how the toolkit facilitates the support for dependability and extensibility. Although home networking applications are usually not performance-sensitive, we measure the overhead of individual system components and present them in the context of end-to-end latencies.

- **Email-based remote control of garage door**

The user rushes out for a meeting and forgets to close the garage door. Sitting in the conference room, he/she sends a signed and encrypted email request to remotely close the garage door. After the email daemon receives, authenticates and parses the request, it locates the garage door opener object through ABLS and NBLS, and instructs the object to close the garage door. To ensure reliable operation, the garage door is equipped with three inexpensive, redundant sensors: a magnetic sensor that detects that the door is at least two inches open, and two horizontal/vertical sensors that detect that the door is at least 25% and 75% open, respectively. The object first reads the states of the sensors from the ABLS to verify that the door is indeed open, after which it sends a powerline control command to achieve the same effect of pushing the garage door opener button. The sensors periodically send out state refresh signals, which are received by the device daemon and converted to ABLS soft-state refreshes. If any of the sensors runs out of battery, its soft-state variable will eventually be timed out so that it does not cause the object to perform incorrect action. Also, a daemon subscribing to sensor variable deletion events will receive an event and display messages on the UAPs to notify the homeowner to change the batteries. To add additional confidence in this critical operation, the object locates the camera in the garage through the lookup services and instructs it to take two snapshots of the garage door, one before the action and one after. The two snapshots are sent back to the user as attachments in the reply email.

On the 550MHz home gateway machine, performance numbers for the various system infrastructure components involved in this scenario for each device operation are: 90ms for the language parser, 85ms for the lookup services (with both ABLS and NBLS running on the gateway machine), and 40ms for the *BindToDisplayName()* operation. The overhead is insignificant compared to the time for the actual device operations (several seconds to 15 seconds) and the time for email delivery.

- **Cell phone-based remote notification of emergency**

The device daemon queries the ABLS for all critical sensors and their corresponding powerline signals, and listens on the powerline, looking for matching signals. A water sensor lies in the crawl space. When it detects water, it sends out a powerline signal, which is translated into a soft-state update by the device daemon (see Section 3.2). Since the email daemon subscribes to the event of changes in any soft-state variable of the CRITICAL_SENSOR type, it receives an event callback from the SSS and sends an emergency email to the homeowner's cell phone email address. It also describes the sensor type and the physical location based on the ABLS information; for example, it says "Crawl space water sensor ON" on the cell phone screen. When a new water sensor is installed in the laundry room and registered in the ABLS, the ABLS change event notifies the device daemon to include the monitoring of signals from the new sensor. Since the soft-state variable associated with the new sensor is of the CRITICAL_SENSOR type, the event subscription of the email daemon does not require any update. This example is a simple demonstration of the extensibility of the Aladdin system.

The elapsed time between the detection of water and the ringing of the cell phone is usually 10 to 40 seconds. Occasionally, it may go up to several minutes, depending on the carrier's text messaging performance. If the gateway machine is not on line and so dialing up is necessary, that would take an additional 40 seconds.

5. Experience Report

The Aladdin system has been running in an actual deployment for several months, and used by the homeowner on a daily basis. For the most part, the system delivers satisfactory services of remote control and notification. However, we have observed problems at several layers, which must be solved before such home networking systems can become mainstream.

The lack of reliability of the X10 powerline control protocol is well known. X10 uses a single frequency to transmit signals and so is particularly sensitive to the signal attenuation problem due to the quality of the powerline wires, and the fluctuation in powerline transmission characteristics as devices are plugged into and removed from outlets throughout the house. Basically, the X10 powerline network is partitioned and the partitioning is dynamic. The soft-state-based distributed system solution provided by Aladdin greatly alleviate the problems by using the phoneline to reach powerline partitions and by allowing the lookup services to adapt to the changes in powerline partitioning. However, it is conceivable that some devices or sensors may become completely isolated from the rest of the powerline, rendering reliable control an impossible task. Next-generation powerline control protocols promise to overcome the above problems at a lower layer and present a reliable broadcast network abstraction over the powerline. Whether and when devices using such protocols can become low-cost consumer-ready devices may be a deciding factor on the broad acceptance of home networking.

Security is another major concern. We have observed in more than one occasion that a faulty

powerline computer interface may exhibit Byzantine behavior by sending random commands over the powerline [WRA+00]. This may create security problems if any of the commands happens to be addressed to a critical device. Even if the random commands only turn on and off less critical devices such as lamps, it creates an extremely unpleasant experience for the user and the diagnosis is not trivial. It remains a challenge for consumer electronics manufacturers to produce low-cost, yet dependable devices. It also requires a system management layer capable of detecting, diagnosing, and, if possible, recovering from anomalies created by faulty devices [WRA+00].

Power outage is an important and interesting failure mode. Since most of the devices in the current Aladdin system rely on electrical power either for operation or for signal transmission, prolonged power outages can potentially shut down the entire system. In the deployment, the gateway machine is connected to a 30-minute UPS and equipped with a power outage sensor. When a power outage occurs and lasts for longer than a few minutes, the email daemon sends out emergency notifications to alert the homeowner to the problem.

Since both remote home automation scenarios involve the use of emails, the Aladdin system is susceptible to email server unavailability and email delivery latency. For example, when the "ILOVEYOU" computer virus/worm and its variants plagued the email systems around the globe, the homeowner lost remote access to his home devices for many hours.

6. Related Work

The concept of soft-states has been widely used in network protocols and distributed systems. But it typically appears in the context of a specific protocol. Our contribution in Aladdin is to identify the importance of soft-states for building robust distributed applications in a heterogeneous and dynamic environment, and to build a soft-state store as a shared infrastructure. The APIs for interacting with the store have been designed to be sufficiently flexible to encompass the existing uses of soft-states in various areas.

In network protocols, soft-states are typically propagated by each node to one or more groups that the node belongs to. Examples include resource reservation protocols such as the receiver-initiated RSVP [ZDE+93] and the sender-initiated YESSIR [PS99], where the soft-states are path states and reservation states. In multicast protocols such as PIM [DEF+96] and SRM [FJM+95], the soft-states are group membership and topology updates. Other examples include periodic route advertisement in routing protocols [DEF+96], directory updates in DNS and in the MBONE Session Announcement Protocol [H96], and data summaries/statistics in transport protocols such as Real-time Transport Protocol [SCF+96] and Soft-State Transport Protocol [RM99]. More generally, we find that most self-stabilizing network protocols essentially use soft-states [AP95].

In distributed systems, soft-states are typically used where strong consistency is not required, but eventual consistency with high availability of information is sufficient. The use of soft-state control information has enabled fault tolerance and high availability in web servers [SLR98], distributed resource management [CRS98], and cluster-based network servers [FGC+97]. Heartbeats or "I am alive" messages have been widely used in detecting failures of remote nodes [HK93][V+98][RMH98]. DCOM pinging [BK98] is a soft-state-based, distributed garbage collection mechanism for server objects to reclaim reference counts associated with abnormally terminated clients.

Jini [E99] provides a set of specifications for services and conventions built atop Java Remote Method Invocation (RMI). Both Jini and Aladdin identified service discovery and lookup, leasing (or soft-states), and events as the most critical infrastructure components in a dynamic distributed system. They differ in the approach of dealing with heterogeneity. Jini focuses on the object-oriented API layer: all network entities must interact with the Jini infrastructure through Java interfaces by either hosting a local Java Virtual Machine or bridging through a Java communication proxy. In contrast, Aladdin specifies the wire protocols for interacting with the infrastructure. Any network entity that is capable of generating messages conforming to the wire protocols can participate in the Aladdin system as a first-class citizen without bridging through a communication proxy. Higher-level object-oriented APIs are provided only for convenience; they simplify the programming on PCs. In another aspect, both Jini and Aladdin allow arbitrary communication protocols between the clients and the services. The difference is that, in Jini, the lookup service entries are Java proxies that provide the interfaces seen by the clients. Any non-Java RMI communication protocols must also be hidden behind these interfaces. In Aladdin, the NBLS entries are opaque address strings. Aladdin is agnostic about the address encoding/decoding process and how a client makes use of an address. This design can facilitate the addition of new protocols and accommodate both RPC and non-RPC style communications. To simplify DCOM programming, Aladdin does provide helper functions for marshaling and unmarshaling DCOM interface pointers. Finally, the concepts of Aladdin Device Adapter and persistent soft-states, the

architecture for remote home automation, and the system management for providing reliable services are orthogonal to Jini and can be applied there as well.

Whether they use soft-states or hard-states, extant lookup services [BvST99, CZH99+, E99] mostly focus on the issue of scalability to millions of users and many groups of lookup services. Being focused on home networks, our lookup services do not emphasize hierarchy or federation. Providing a lightweight solution to support friendly naming of devices has been the key requirement. The NBLS wire protocol uses similar semantics to an early version of the Simple Service Discovery Protocol (SSDP) [G+99], which is part of the Universal Plug and Play (UPnP) initiative. In general, Aladdin provides higher level services for home networking and can build on top of UPnP. Since eventual consistency based on soft-state refreshes is sufficient for most home networking applications, we did not adopt active replication-based solutions [M96] that use totally ordered, reliable multicast to achieve virtual synchrony across replicated lookup services, thereby guaranteeing strong consistency.

Scalability of the soft-state networking protocols necessitates dealing with the message overhead introduced by the periodic refresh of information. This has led to several refinements: batching and compression of refreshes between nodes [WTZ99], adaptation of refresh frequency (either by negotiation between senders and receivers or independently by senders and receivers by monitoring control traffic) [SEF+97], use of acknowledgements for refreshes [RM99], and multistage decrease of frequency of refresh [PH97]. Our toolkit accommodates most of these refinements. It uses acknowledgements albeit only for low-frequency refreshes which benefit the most from acknowledgements in the presence of message loss.

Extant research projects on smart home environments are largely focusing on issues orthogonal to dependability and extensibility. Research results from those projects can be incorporated into the Aladdin environment to further enhance overall user experience. The Adaptive House project [M98] aims at developing a home control system that observes the lifestyle of the inhabitants and learns to anticipate their needs based on neural network techniques. The goal of the EasyLiving project [S+98] is to build an intelligent environment that maintains an awareness of its occupants through computer vision and facilitates unencumbered interactions among people and devices. The Aware Home project [Es99] focuses on analyzing and interpreting captured sensor streams from high-end multi-modal sensors to make the environment aware. The Smart Floor project [O00] has created a system for identifying people based on their footstep force profiles.

7. Summary

We have shown that the combination of soft-state store, publish/subscribe eventing, attribute-based lookup service, and name-based lookup service provides a powerful programming abstraction for building dependable and extensible home networking applications. To provide dependability, the soft-state store serves as a uniform framework for detecting the failures and unavailability of devices and objects, and for removing stale data to reclaim system resources as well as preventing misuse of such data. The publish/subscribe event system further allows higher-level adaptive, corrective, alerting, and cleanup actions to be performed in response to changes in the soft-state store. The late-binding feature provided by the lookup services makes the system more robust in the presence of failures, recoveries, and mobility. To provide extensibility, the attribute-based and name-based lookup services allow devices and objects to join dynamically and be available to future clients. Event subscriptions based on soft-state types, instead of individual variables, further allow triggering existing clients to include the new devices and objects. We have described two remote home automation applications and summarized our experiences from the actual usage of these applications.

Acknowledgement

We thank Kuansan Wang (Microsoft Research) for providing the language parser and speech recognizer.

References

[AP95] A. Arora and D. Poduska, "A Timing-based Schema for Stabilizing Information Exchange in Networks," in *Proc. Int. Conf. on Computer Networks*, 1995.

[B98] D. Box, Essential COM, Addison Wesley, 1998.

[BvST99] G. Ballatijn, M. van Steen, A. Tannebaum, "Exploting location awareness for scalable location-independent object IDs," in *Proc. Fifth Annual ASCI Conf.*, pp. 321-328, 1999.

[BK98] N. Brown and C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, 1998.

[CRS98] K. Chandy, A. Rifkin, and E. Schooler, "Using announce-listen with global events to develop distributed control systems," *Concurrency: Practice and Experience*, pp. 1021-1027, 1998.

[C88] D. D. Clark, "The design philosophy of the DARPA internet protocols, in *Proc. ACM SIGCOMM*, 1988.

[CM] CM11A Programming Specification, ftp://ftp.x10-beta.com/ftp/protocol.txt.

[CZH99+] S. Czerwinski, B. Zhao, T. Hodes, et al, "An architecture for secure service discovery service", in *Proc. ACM/IEEE MobiCom*, pp. 24-35, Aug. 1999.

[DEF+96] S. Deering, D. Estrin, D. Farinacci, et al, "PIM architecture for wide-area multicast routing," *IEEE/ACM Trans. on Networking,* Vol. 4, No. 2, pp. 153-162, Apr 1996.

[E99] W. K. Edwards, "Core Jini", Prentice-Hall Inc., 1999.

[Es99] I. Essa, "Ubiquitous Sening for Smart and Aware Environments: Technologies towards the building of an Aware Home," *Position paper for the DARPA/NSF/NIST Workshop on Smart Environments,* July 1999.

[FJM+95] S. Floyd, V. Jacobson, S. McCanne, et al., "A reliable multicast framework for light-weight sessions and application level framing," *Proc. SIGCOMM*, Sept. 1995.

[FGC+97] A. Fox, S. Gribble, Y. Chawathe, et al, "Cluster-based scalable network services," in *Proc. SOSP,* pp. 78-91, 1997.

[G+99] Y. Y. Goland et al., "Simple Service Discovery Protocol," *IETF Internet Draft*, http://www.ietf.org/internet-drafts/draft-cai-ssdp-v1-03.txt, Oct. 1999.

[H96] M. Handley, "SAP: Session Announcement Protocol," Internet Draft, Internet Engineering Task Force, Nov. 19, 1996

[H98] The Home Phoneline Networking Alliance, "Simple, High-Speed Ethernet Technology for the Home," http://www.homepna.org/docs/wp1.pdf, June 1998.

[HF95] D. Haught and J. Ferguson, "Jet Database Engine Programmer's Guide," Microsoft Press, 1995.

[HK93] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," in *Proc. FTCS*, pp. 2-9, 1993.

[M96] S. Maffeis, "A Fault-Tolerant CORBA Name Server," in *Proc. SRDS*, pp.188-197, Oct. 1996.

[M98] M. C. Mozer, "The Neural Network House: An Environment that adapts to its inhabitants," in *Proc. AAAI Spring Symp. on Intelligent Environments*, pp. 110-114, 1998.

[O00] R. J. Orr and G. D. Abowd, "The Smart Floor: A Mechanism for Natural User Identification and Tracking," *Technical Report GIT-GVU-00-02,* Georgia Tech., Jan. 2000.

[PH97] P. Pan and H. Schulzrinne, "Staged refresh timers for RSVP," in *Proc. GLOBECOM*, Vol. 3, pp. 1909-1913, 1997.

[PS99] P. Pan and H. Schulzrinne, "YESSIR: a simple reservation mechanism for the Internet," *Computer Communication Review*, Vol. 29, No. 2, pp. 89-101, 1999.

[RM99] S. Raman and S. McCanne, "A model, analysis, and protocol framework for soft state-based communication," in *Proc. SIGCOMM,* pp. 15-25, 1999.

[S98] Silent Servant Home Control Inc., Automated Home Control, 1998.

[SCF+96] H. Schulzrinne, S. Cassner, R. Frederick, et al, "RTP: A transport protocol for real-time applications", *IETF Audio Visual Transport Working Group, RFC-1889*, Jan. 1996.

[S+98] S. Shafer, J. Krumm, B. Brumitt, B. Meyers, M. Czerwinski, and D. Robbins, "The New EasyLiving Project at Microsoft Research," *DARPA/NIST Workshop on Smart Spaces*, July 1998.

[SEF+97] P. Sharma, D. Estrin, S. Floyd, et al, "Scalable timers for soft state protocols," in *Proc. INFOCOM*, pp. 222-229, 1997.

[SLR98] A. Singhai, S.-B. Lim, and S. R. Radia, "The SunSCALR Framework for Internet Servers," in *Proc. FTCS,* pp.108-117, 1998.

[RMH98] R. van Renesse, Y. Minsky, and M. Hayden, "A Gossip-Based Failure Detection Service," in *Proc. Middleware,* 1998.

[V+98] W. Vogels et al., "The Design and Architecture of the Microsoft Cluster Service," in *Proc. FTCS*, pp. 422-431, 1998.

[WRA+00] Y. M. Wang, W. Russell, A. Arora, et al., "Towards Dependable Home Networking: An Experience Report," in *Proc. IEEE Int. Conf. on Dependable Systems and Networks (formerly FTCS),* June 2000.

[WTZ99] L. Wang, A. Terzis, and L. Zhang, "New proposal for RSVP refreshes," in *Proc. 7th Int. Conf. on Network Protocols (ICNP'99)*, pp. 163-172, 1999.

[ZDE+93] L. Zhang, S. Deering, D. Estrin, et al, "RSVP: a new resource ReSerVation Protocol ," *IEEE Network,* Vol. 7, No. 5, pp. 8-18, Sept. 1993.

WSDLite: A Lightweight Alternative to Windows Sockets Direct Path[‡]

Evan Speight
Computer Systems Laboratory
Cornell University
espeight@csl.cornell.edu

Hazim Shafi
Trilogy Software, Inc.
Austin, Texas
hazim_shafi@trilogy.com

John K. Bennett
Department of Computer Science
University of Colorado at Boulder
jkb@cs.colorado.edu

Abstract

This paper describes WSDLite, a thin software layer that maps a useful subset of the WinSock2 API onto a system area network. The development of WSDLite was motivated by our experience with an early version of Windows Sockets Direct Path (WSDP). WSDP was developed by Microsoft to allow unmodified network applications to exploit the performance and reliability advantages of System Area Networks (SANs). This is accomplished through the use of a software "switch" that, when appropriate, redirects message traffic through the SAN provider protocol stack instead of the standard TCP/IP protocol stack. In addition to the performance advantages, the WSDP architecture offers several other benefits, including automatic support for legacy code, a single well-known API for supporting many different underlying SAN network protocols, and substantially simpler buffer management than that required by the native SAN API. The beta version of WSDP that we examined did not perform as well as expected, achieving only 26% of the native SAN throughput on the system studied. In an effort to determine whether or not this performance difference was intrinsic, we developed WSDLite, a simple alternative to WSDP. WSDLite is a user-level runtime library that implements a small but commonly used subset of the WinSock2 API. For those applications that do not require full WinSock2 functionality, WSDLite provides both the transparency of WSDP and much of the performance benefit of the underlying SAN architecture. In low-level network tests, WSDLite achieves an average of 70% of the native SAN performance. In this paper we describe the design of WSDLite, and present results comparing the performance of both parallel applications and low-level benchmarks using WSDLite, WSDP, TCP, and a native SAN programming library API as the network programming layer.

1. Introduction

System area networks (SANs) are characterized by high bandwidth; low latency (on the order of 10μsec or less for zero-length messages); a switched network environment; reliable transport service implemented directly in hardware; no kernel intervention to send and receive messages; and little or no copying on either the sending or receiving side. SANs may be used for enterprise applications such as databases, web servers, reservation systems, and small to medium scale parallel computing environments.

System area networks have not yet enjoyed wide adoption, in part because of the difficulty associated with writing applications to take advantage of network programming libraries that generally ship with SAN hardware. In order to provide low latency, zero or single-copy messaging between nodes in a SAN, programmers must address a variety of buffer management and flow control issues not typically associated with TCP/IP-style network programming. These issues stem primarily from the use of DMA between the network interface card and the host memory, a process that allows system area networks to provide orders-of-magnitude lower latencies and lower processor utilizations than previous network architectures and protocols. Addressing these requirements can represent a significant burden, not only to programmers developing new applications, but also to those who wish to obtain the benefits of system area networks for the many millions of lines of existing network application code.

To address these concerns, Microsoft, working with SAN implementers, has developed an alternative that will allow network applications to obtain many of the performance benefits associated with system area networks while retaining the familiar programming interface of Berkeley-style sockets in the WinSock2 API. This technology, called *Windows Sockets Direct*

[‡]This research was conducted while all authors were members of the Department of Electrical and Computer Engineering, Rice University, Houston, Texas.

Path (WSDP) [4], fits immediately below the network application and routes network communication calls to either the standard TCP/IP protocol stack or to the WinSock SAN Provider stack, which utilizes the SAN's native network communication mechanism to achieve low latency, high throughput messaging. One of the principal benefits of WSDP is that existing WinSock2-compliant applications do not have to be rewritten, or even recompiled. Currently, WSDP is restricted to use with the Data Center version of the Windows 2000 operating system.

WSDP necessarily implements the entire WinSock2 API, and as a result, incurs overhead costs associated with providing full functionality. In the beta version of WSDP that we have examined, this overhead is quite substantial. While we expect release versions of WSDP software to exhibit better performance than the current beta version, we also believe that there are attractive design alternatives for those applications that do not require full WinSock2 functionality. This paper explores one such alternative.

We have implemented WSDLite, a protocol layer that implements a subset of the WinSock2 API on top of the raw programming interface provided by the GigaNet cLAN implementation of the Virtual Interface Architecture (VIA) [3]. The VI Architecture is the proposed standard for user-level networks developed by Microsoft, Compaq, and Intel. The cLAN architecture provides 9µsec latency for zero-byte messages in our system area network environment when using the VI Programming Library (VIPL) API. WSDLite, similar to WSDP, allows programs written to use TCP/IP to obtain the performance benefits associated with an underlying network architecture that supports VIA. We make use of the Detours [9] binary rewriting software package to intercept the TCP calls implemented by WSDLite and route them to the WSDLite implementation of these functions, while forwarding TCP calls not implemented within WSDLite to the standard WinSock2 protocol stack. Detours allows us to run Winsock2-compilaint applications without recompilation. Unlike WSDP, however, WSDLite only implements a subset (approximately 10%) of WinSock2 functions. The functions implemented were chosen based upon their common use in a variety of software available at our site. A lighter-weight protocol layer such as WSDLite can provide substantial performance benefit relative to full-functioned protocol layers for applications that do not need the full TCP/IP functionality provided by WSDP. Additionally, WSDLite can be used on any Windows NT or Windows 2000 system for which VIA support is available; it is not restricted to Windows 2000 Data Center. We have successfully tested WSDLite on clusters comprised of Windows NT 4.0 workstations and servers, Windows 2000 Professional Workstations, and Windows 2000 Data Center Servers. Simple network latency tests show WSDLite to be an average of 59% faster than the beta WSDP implementation across all message sizes up to 32 Kbytes.

We examine the performance of WSDLite using several network benchmark programs. First, we compare the performance of a series of low-level benchmarks with (1) TCP/IP using WinSock only, (2) TCP/IP using WSDP, (3) TCP/IP using WSDLite, and (4) a version written to use the native VIPL API. For each of the low-level benchmarks, we report roundtrip latency and network throughput. We also report processor utilization, as well as throughput per CPU second, which brings into focus the tradeoff between network and application performance. We next examine the overhead associated with the use of the Detours [9] library to provide Winsock2 transparency. Finally, we use the same four network layer implementations as the messaging layer for the Brazos Parallel Programming Library. By running a set of parallel applications utilizing Brazos, we can evaluate the performance of each network alternative on real applications.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the Virtual Interface Architecture in order to provide the context for the discussion of Windows Sockets Direct Path in Section 3. Section 4 describes the design and implementation of WSDLite. In Section 5 we report the results of our experimental comparison of WSDLite and WSDP. Related work is described in Section 6. We conclude and discuss future work in Section 7.

2. Overview of the VI Architecture

Although Windows Sockets Direct Path is designed to work with a variety of system area network architectures, we are only aware of current WSDP support in the context of the Virtual Interface Architecture. In this section, we present an overview of the VI Architecture as implemented on the GigaNet cLAN GNN1000 network interface card.

Figure 1 depicts the organization of the Virtual Interface Architecture. The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider consists of the VI Network Adapter and a Kernel Agent device driver. The VI Consumer is composed of an application program and an operating system communication facility such as MPI or sockets, although some "VI-aware" applications communicate

directly with the VI Provider API. After connection setup by the Kernel Agent, all network actions occur without kernel intervention. This results in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for creation/destruction of VI's, VI connection setup and teardown, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

overhead associated with traditional network protocol stacks, the VI Architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider. This registration process locks down the appropriate pages in memory, which allows for direct DMA operations into user memory by the VI hardware, without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided

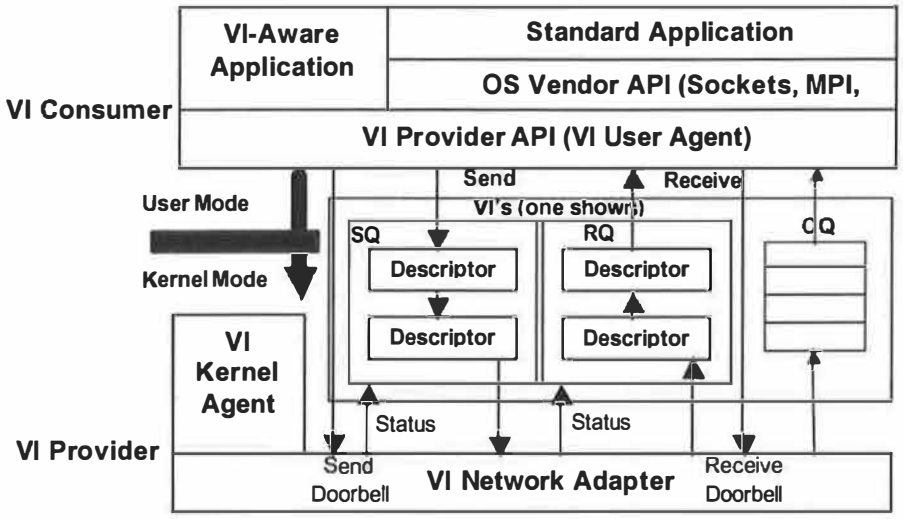


Figure 1. Block Diagram of the Virtual Interface Architecture

A VI consists of a Send Queue and a Receive Queue. VI Consumers post requests (Descriptors) on these queues to send or receive data. Descriptors contain all of the information that the VI Provider needs to process the request, including pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them when completed. VI Consumers remove completed Descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated "Doorbell" that is used to notify the VI network adapter that a new Descriptor has been posted to either the Send or Receive Queue. The Doorbell is directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of Descriptor completions of multiple VI's without requiring an interrupt or kernel call.

2.1. Memory Registration

In order to eliminate the copying between kernel and user buffers that accounts for a large portion of the to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers, thereby avoiding duplication of locking and translation operations. Memory registration also takes page-locking overhead out of the performance-critical data transfer path.

2.2. Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send and receive semantics, and direct reads and writes to and from the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote node (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. The VI Consumers on both the sending and receiving nodes specify the location of the data. On the sending side, the sending process specifies the memory regions that contain the data to be sent. On the receiving side,

the receiving process specifies the memory regions where the data will be placed. The VI Consumer at the receiving end must post a Descriptor to the Receive Queue of a VI before the data is sent. The VI Consumer at the sending end can then post the message to the corresponding VI's Send Queue.

Remote DMA transfers occur using the same descriptors used in send/receive style communication, with the memory handle and virtual address of the remote memory specified in a second data segment of the descriptor. VIA-compliant implementations are required to support remote write, but remote read capability is an optional feature of the VIA Specification. The GigaNet cLAN architecture only provides for remote writes.

3. Windows Sockets Direct Path

Windows Sockets Direct Path (WSDP) allows programs written for TCP/IP to transparently realize the performance advantages of user-level networks such as VIA. Programs developed to the WinSock2 API do not have to be rewritten to take advantage of changes in underlying network architecture to a SAN, nor is recompilation of these programs necessary. This enables legacy network code to work "out of the box" and enjoy at least some benefit of the low message latency associated with SANs. Although WSDP is designed to work with a variety of low-latency SAN architectures, we restrict our discussion here to how WSDP interacts with the cLAN VIA architecture described in Section 2.

WSDP removes many of the pedantic tasks that must be addressed by programs that directly access the VIPL API. These include memory registration, certain aspects of buffer management, and the effort required to port and recompile a sockets-compliant application to use the VIPL API. In the following sections we describe the basic technology associated with WSDP as well as some programming considerations that must be addressed to use WSDP effectively.

Figure 2 depicts a block diagram of the WSDP architecture. The key component of the WSDP architecture is the software switch, which is responsible for routing network operations initiated by WinSock2 API calls to either the standard TCP/IP protocol stack, or to the vendor-supplied SAN WS Provider. In addition to providing access to both of these pathways to the network on an operation-by-operation basis, the switch provides several important functions through the use of a lightweight session executed on top of the SAN provider. This session provides OOB (out of band) support, flow control, and support for the *select* operation. None of these mechanisms are traditionally provided by a typical SAN architecture. There are several operations that require the support of the TCP/IP protocol stack (i.e., do not use WSDP), including:

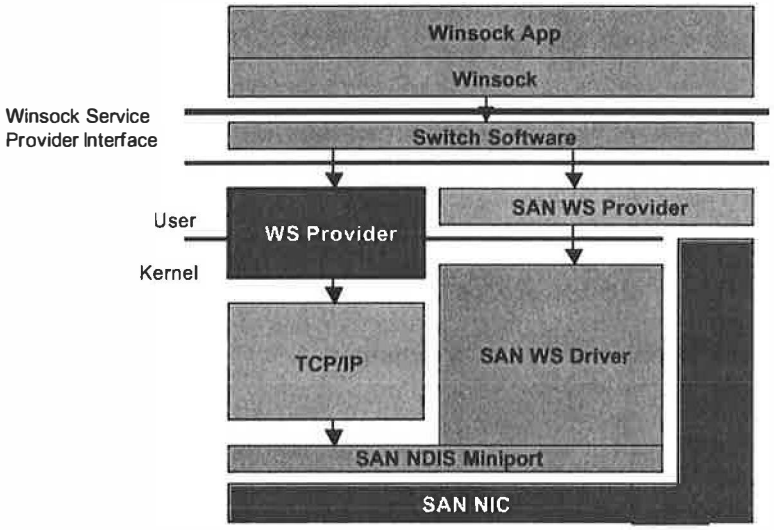


Figure 2. Switch Architecture for Windows Sockets

- Connections to remote subnets.
- Socket creation.
- Raw sockets and UDP sockets - Because SANs support connection-oriented reliable communication all connectionless and

uncontrolled communication must be handled by the TCP/IP protocol stack. This limits the applicability of WSDP to those applications that (a) use TCP, and (b) do not make use of group communication.

In addition to these restrictions on the use of WSDP, system calls are required to complete most overlapped I/O calls, increasing the latency of these calls due to induced operating system overhead.

The switch component is also responsible for taking care of several programming details that usually must be addressed by the programmer writing directly to the programming library supplied with SANs. A brief discussion of these details follows:

- Buffer registration – As discussed in Section 2.1, buffer space used for messaging must be registered with a SAN provider in order to allow direct DMA into and out of host memory by the NIC. However, there is no provision for this functionality in the WinSock2 specification, as the operating system handles message buffering through copying in a standard WinSock environment. Therefore, the switch component is responsible for ensuring that all buffer regions used for communication are registered with the SAN provider prior to use.

- Buffer placement – Another issue relating to the management of buffers in a system area network requires there to be a buffer posted to a network endpoint prior to receipt of an incoming message. This is again related to the use of DMA between the network interface card and the host memory and the lack of flow control associated with SAN NICs. The switch software pre-posts small buffers to each connection opened through the WS SAN Provider in order to handle incoming messages.

- Support for RDMA – Most system area network include support for remote memory operations, allowing a host node to directly write and/or read data directly from a remote node's address space. No such API exists in the WinSock2 specification. WSDP makes use of the remote write capability of the cLAN architecture in a manner similar to that of WSDLite, as discussed in the next section.

4. WSDLite

WSDLite implements approximately 10% of the WinSock2 API. The following functions are currently implemented by WSDLite: **WSAStartup()**, **WSACleanup()**, **WSASocket()**, **socket()**, **connect()**, **listen()**, **accept()**, **bind()**, **send()**, **WSASend()**, **recv()**, **WSARecv()**, **select()**, **closesocket()**, and **WSAGetLastError()**.

When an application calls a function supported by WSDLite, the function call is intercepted by the Detours [9] runtime library and redirected to the version of the function implemented by WSDLite. In order to leverage functionality existing in the WinSock TCP/IP protocol stack that is not directly related to messaging performance (such as connection procedures and name resolution), some of the WSDLite functions make calls to their WinSock counterparts from within the WSDLite library. For instance, during connection procedures, the WSDLite implementation of **bind()** calls the WinSock2 version of **bind()** internally to check for errors such as two sockets being bound to the same port. In fact, WSDLite duplicates the entire connection process internally on the default TCP/IP protocol stack in order to catch such errors, greatly reducing the code size of the WSDLite implementation.

4.1. Sending Data in WSDLite

When a message is to be sent on a connected pair of sockets, the WSDLite implementation of **WSASend()** or **send()** first must register the buffer containing the data to be sent, if it is not already registered with the cLAN NIC.

Memory Registration Issues

Registering memory is an expensive operation for two reasons. First, registering and deregistering memory on each network access would add unacceptable latency to network operations, especially for small messages. We measured the cost of registering memory for buffer sizes up to 32 Kbytes, and found that it takes roughly 15 µsec to register and deregister a region of memory with the VI Provider, regardless of buffer size. This time increases linearly with buffer size after the size exceeds the 64K segment size used by the NT virtual memory manager. To address this issue, WSDLite maintains a hash table of address ranges that have been used as messaging buffers previously, and this table is consulted before a message can be sent. There are three possible outcomes from the initial hash table lookup:

1. The address has previously been registered, and the size registered is equal or larger than the size of the buffer currently posted. No other action is required.
2. The address has been previously registered, but the size of the region registered does not encompass the entire buffer currently posted. The currently registered region must be deregistered with the NIC and the new region registered.
3. The address has not been previously registered, and WSDLite must register the entire buffer.

To reduce the amount of registering that must be performed by WSDLite, it is important for application programmers to reuse buffers as much as possible.

The second source of overhead associated with memory registration results from the fact that a part of the memory registration process involves pinning messaging buffers into physical memory, which may reduce the resources available for applications. To address this problem, WSDLite employs a simple garbage collection scheme based on timestamps to reclaim unused message buffer space before the amount of pinned RAM impacts application performance.

Choosing the Correct Send Semantic

We have found that minimum latency for messages may be obtained in one of two ways, depending on the size of the message. For small messages, the best performance is achieved by copying data out of temporary receive buffers into the application buffers posted by the corresponding receive operation. For larger messages, lower latency can be achieved by taking advantage of VIA's RDMA capability. When a large message is to be sent, the sending process first sends a setup message to the receiver. This message contains the length of the message to be sent. The receiver registers the memory region to be received into (if it is not already available), and then returns the virtual address and memory region handle to the sending process. The sending process then remote-writes the data directly into the address space of the receiving process, and sends a completion message containing the size of the message written to the receiver when the operation has completed.

The message size at which WSDLite switches from memory copying to RDMA depends on the speed of the host processors, the efficiency of the memory hierarchy, and the latency of network operations.

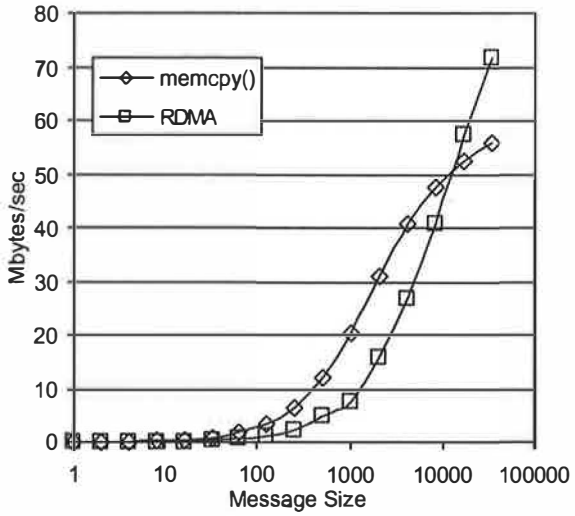


Figure 3. Bandwidth Crossover Point

The crossover point can be clearly seen in Figure 3, which shows the sustainable bandwidth of WSDLite when copying is always used regardless of message size (labeled **memcpy()**), and when RDMA is always used. In the case of our system, the crossover point occurs between 8K and 16K. More precise measurements pinpoint it at 11.9 Kbytes. In general, if copying a memory region of size *n* takes less time than the two additional small messages necessary for the RDMA transfer, memory copying will achieve better performance. Because this value is likely to be different on different machines, WSDLite attempts to automatically determine the optimum value for this cutoff the very first time a socket is created. When the first socket on a machine is created, a small test is run that measures the time to copy regions of memory of varying sizes. When a connection is first made to a remote machine, a test to determine the latency of message sizes corresponding to the setup and acknowledgement messages required for RDMA transfer is also run. The cutoff point for this particular machine can then be determined, and this value is stored in a registry entry that is consulted each time an application makes a connection through WSDLite to a specific remote machine. This step only occurs once during the connection to a remote machine. Subsequent network programs that connect to the remote machine can simply retrieve the cutoff value from the registry based on the remote machine to which the connection is being made. The registry value may be deleted by an administrator at any time to force a recalculation of this parameter, or overridden manually.

4.2. Choice of WSDLite Functions

Finally, we conclude this section with a brief discussion on the functions that we chose to implement in WSDLite. We implemented only those calls that provide the network functionality required by our suite of network programs used for this evaluation. We believe these to be representative of a larger class of network applications that only use basic TCP functionality. By keeping the number of functions small, and the implementation thin, we are able to realize a high percentage of the performance available from the SAN. Many other WinSock2 functions could easily be added to the WSDLite implementation by using our initial functions as a starting point. The downside to our strictly user-level approach is that a different version of WSDLite must be used for each SAN network programming library. However, precisely because we have kept the number of functions both small and basic, this is not a difficult thing to do. The approach taken by WSDP, on the other hand, is one of providing full functionality regardless of the underlying SAN network. This implies that 1) many functions, whose implementations may not easily map to the SAN programming API, will have high overhead; and 2) another level of indirection must exist between the switch software provided by Microsoft and the hardware vendor-provided SAN layer. These two observations necessitate an implementation with higher overhead than a simple user-level library such as WSDLite. Therefore, WSDLite is proposed as a performance alternative to WSDP in certain situations, not a replacement for applications requiring full TCP functionality.

5. Experimental Results

In this section we begin by describing our experimental platform. We then present results comparing several important low-level network performance measurements run under WSDP, WSDLite, TCP, and VIPL on two uniprocessor nodes. Next, we discuss these same measurements when SMP nodes are used. Finally, we conclude the section with results showing the performance of four scientific parallel applications using the four network layer alternatives when run on a larger cluster of SMP servers.

5.1. SAN Configuration

All experiments were performed using a cluster of Compaq Proliant 6400 servers running the Beta 2 release of Windows 2000 Data Center Server, build 2195. Each machine contains one to four 500 Mhz Pentium-III processors, 512 Mbytes of SDRAM, and dual 64-bit PCI busses running at 66 Mhz. The interconnection network is implemented with a single GigaNet GNN1000 NIC in each machine connected via a GNX5000 switch. The switch cut-through latency is 580 ns. The unidirectional latency for a zero-byte message on this system is 9 µsec, and the peak sustainable bandwidth that we have observed is 102 Mbytes/sec.

5.2. Low Level Results

In this section we compare the performance of a message ping-pong test that simply sends messages between two nodes in the cluster. Each node waits for a reply before sending the next message. We compare the performance of this test when using WSDLite, the TCP/IP protocol stack shipped with Windows 2000, WSDP, and the same test written directly to the VIPL API. Note that the first three tests are the same executable; no modifications were necessary when using WSDLite or WSDP to take advantage of the underlying VI hardware. We examine the performance of each of these schemes for message sizes up to 32Kbytes with respect to roundtrip latency, peak sustainable bandwidth, processor utilization, and Mbytes/CPU-second. Finally, we look at the overhead associated with using the Detours [9] package to provide transparent access to WSDLite through the WinSock2 API. Results in this section have been obtained with a single processor in each of the two machines being used. The results of making the same measurements with four processors in each machine is discussed in Section 5.3.

Figures 4 and 5 show the performance of our ping-pong test as measured by roundtrip latency and peak sustainable bandwidth for message sizes from 1 byte to 32 Kbytes. With a single processor in each system, we see that the latency of WSDLite is on average only 19.2% higher than that of native VIPL across all message sizes. The differences between WSDLite and VIPL stem from the extra overhead on each network call of traversing through the TCP-to-VIPL translation layer, the overhead associated with trapping WinSock2 calls using Detours, and the buffer management and flow control that WSDLite must implement.

As expected, TCP performs poorly on latency and peak bandwidth measurements with respect to either WSDLite or VIPL. WSDP performs similarly to TCP, but actually has higher latency at all message sizes and averages 28.8% higher than TCP. The performance of WSDP lags that of WSDLite by an average of 67.9% for all message sizes. This performance advantage of WSDLite is slightly higher at smaller message sizes,

with a 69.5% improvement for single-byte messages and a 59.1% improvement for 32Kbyte messages.

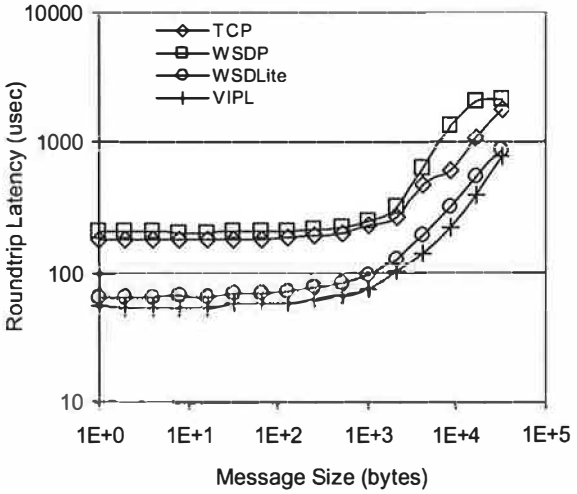


Figure 4. Roundtrip Latency

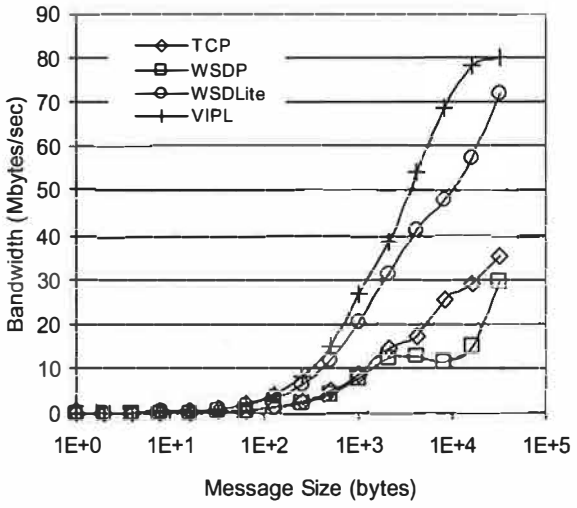


Figure 5. Peak Sustainable Bandwidth

Figure 5 shows that the bandwidth of TCP and WSDP peak at a maximum of around 30-35 Mbytes/sec, whereas VIPL achieves nearly 80 Mbytes/sec, and WSDLite around 72 Mbytes/sec. The performance of WSDLite is restricted below the 16 Kbyte message size from additional copying out of the pre-posted receive buffers, and from the extra setup and acknowledgement messages necessary to implement the RDMA transfer at 16 and 32 Kbyte message sizes. However, these overheads still allow WSDLite to perform within 22% of VIPL. The significantly higher overheads of WSDP caused by multiple software layering and polling between these layers results in performance that is worse than just using TCP directly, regardless of message size.

Figure 6 shows the average processor utilization for the uniprocessor execution of our ping benchmark. For small messages, VIPL has a much higher processor utilization than either of the other three implementations, resulting from a time compression effect due to the small amount of time the message requires "on the wire", and the small fixed costs due to the low overhead of the network protocol. WSDLite and TCP display similar utilizations at small message sizes due to their higher fixed-cost overhead relative to VIPL. WSDP shows the lowest overall utilization for message sizes less than 1K. All implementations that use VI in some layer (WSDP, WSDLite, and VIPL) show low processor utilizations at large message sizes due to the fact that large messages require relatively long DMA times to transfer the message to the NIC hardware, during which time the processor is idle. TCP, on the other hand, buffers and copies messages internally, keeping the utilization high throughout the entire range of message sizes.

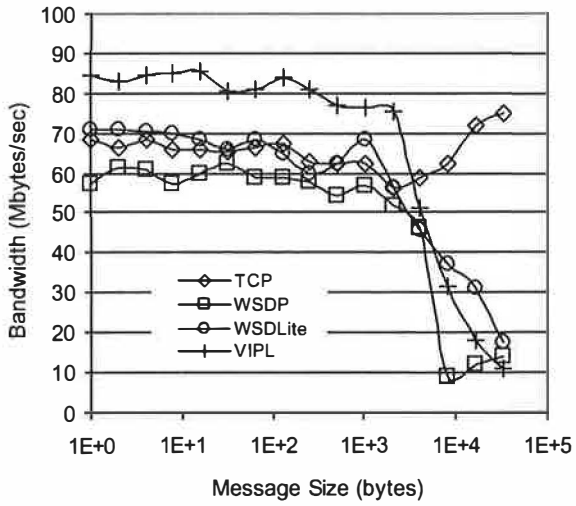


Figure 6. Processor Utilization

The data presented in Figure 6 is misleading, seeming to indicate that WSDP is the most efficient protocol because the processor utilization is lower at smaller message sizes, and the VI Architecture was designed to maximize the performance of small messages [4]. By dividing the peak bandwidth achieved (as presented in Figure 5) by the processor utilization necessary to sustain this bandwidth (as shown in Figure 6), we can track the relative efficiency of a particular network protocol or architecture and find

out how much processing time is required to send a fixed amount of data. Figure 7 shows this measurement for the ping test using TCP, WSDP, WSDLite, and VIPL, and is expressed in Mbytes/CPU-second. With only a single processor, TCP and WSDP perform particularly poorly using this metric at small message sizes. The relatively low processor utilization displayed by WSDP in Figure 6 is offset by the extremely low network throughput shown in Figure 5, causing WSDP's performance to nearly mirror that of TCP for message sizes below 8Kbytes.

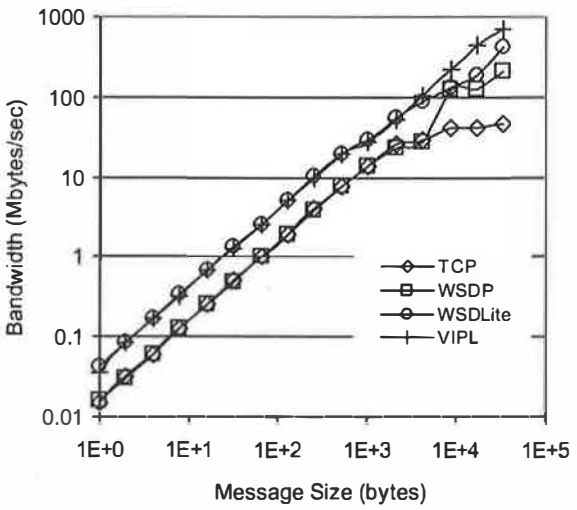


Figure 7. Messaging Efficiency

WSDLite and VIPL, on the other hand, more than make up for the additional processor utilization required with improved bandwidth. Both protocols perform similarly at small message sizes because of the low overhead imposed by the runtime system. This results in a higher amount of data transferred per processor cycle than either WSDP or TCP. As message size increases and the fixed "wire time" becomes a larger portion of the overall network time, the Mbytes/CPU-seconds metric for all protocols increases as the processor overhead becomes less of a factor in overall performance. For very large messages, the performance of the three architectures that utilize VIA begin to converge, whereas the high processor utilization causes the TCP performance to flatten out between 8 and 32 Kbytes.

5.3. SMP Performance

In order to evaluate the performance benefits of running each implementation in an SMP environment, we repopulated each of the two machines used in the experiments with four 500 MHz P-III processors. The performance difference between these results and those presented in Section 5.2 stem from the level of concurrency exploited by the runtime system, as well as the overhead associated with managing threads residing on different processors. Table 1 shows the thread counts present in each process during the execution of the test. Note that the thread counts did not change when moving from a uniprocessor to a 4-way SMP.

	# Threads	Thread Breakdown
TCP/IP	2	1 user thread 1 Winsock thread
WSDP	7	1 user thread 6 WSDP threads
WSDLite	3	1 user thread 1 Winsock thread 1 VIPL thread
VIPL	2	1 user thread 1 VIPL thread

Table 1. Thread Usage

From the thread counts shown in Table 1, we would expect WSDP to exploit concurrency and thus show an improved performance with multiple processors. The other three architectures do not use concurrency in an attempt to reduce overhead. With respect to peak bandwidth, we found that WSDP does indeed perform better with SMP nodes by an average of 17% across all message sizes. The largest improvement occurred at 16K messages (48%). Because neither WSDLite nor VIPL use concurrency within the runtime system, the performance of these two implementations remains nearly constant regardless of the number of processors available (average improvement of 4.5% and .6%, respectively). However, the throughput of WSDLite remains an average of 67% better than that of WSDP across all message sizes.

5.4. Overhead Associated with Detours

Finally, we examine the performance impact of using Detours to eliminate the necessity of recompiling a WinSock2 application to use the WSDLite library. Detours instruments x86 binaries and inserts jump calls to trap targeted Win32 functions. We have configured Detours to trap all of the calls implemented in the WSDLite library and redirect them to the WSDLite implementation of the functions. Using this redirection, as opposed to recompiling the program and linking directly with the WSDLite library, incurs a 3 µsec overhead per roundtrip message. It may be possible to reduce this further through more direct interception methods.

5.5. Application Results

In this section we examine the performance of WSDP, WSDLite, TCP, and VIPL when each is used as the underlying network layer for the Brazos Parallel Programming Environment [12]. Brazos provides transparent shared memory and message passing support across a network of SMP machines running Windows 2000/NT. Brazos was originally developed for use with UDP on WinSock, delivering superior performance to distributed shared memory applications. Support for VI was later added. For the purposes of this study, we converted Brazos to run using TCP sockets and present results for four shared memory scientific applications running on two quad Compaq Proliant 6400 servers. The four applications include Ilink, a genetic linkage program used to trace genes through family histories; Barnes Hut, an n-body problem solver from the SPLASH-2 benchmark suite [15]; LU decomposition, also from SPLASH-2; and FFT-3D, used to solve fast Fourier transforms in three dimensions, from the NAS parallel benchmark suite [1].

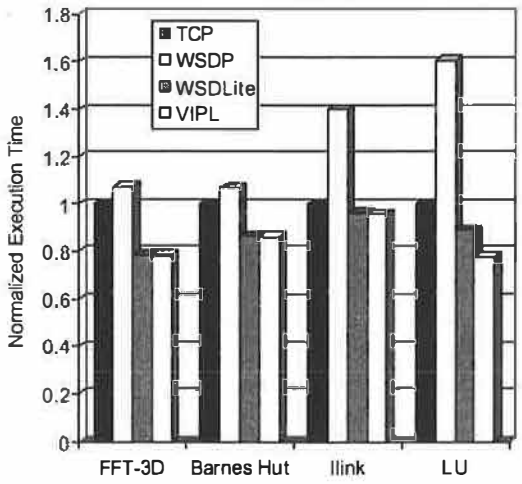


Figure 8. Parallel Application Performance

Figure 8 shows the performance of these four applications on WSDLite, WSDP, and VIPL in terms of the execution time normalized to that of the execution time when run on TCP. For three of the four applications (FFT-3D, Barnes Hut, and Ilink), WSDLite performs within 2% of the VIPL performance, demonstrating the low overhead associated with the WSDLite runtime protocol layer. For LU, the performance of WSDLite suffers slightly due to send throttling in the WSDLite protocol, causing some send operations to stall waiting for available buffers to be re-posted on the receiving node.

For these experiments, WSDP performs particularly poorly relative to TCP. We believe this performance degradation to be the result of processor contention due to the high number of threads used in the WSDP protocol stack (see Table 1). Threading in WSDP is used to boost concurrency between the software layers that make up the protocol stack. Synchronization and polling between these layers apparently results in processor starvation for computation threads, leading to a potentially large increase in parallel execution time.

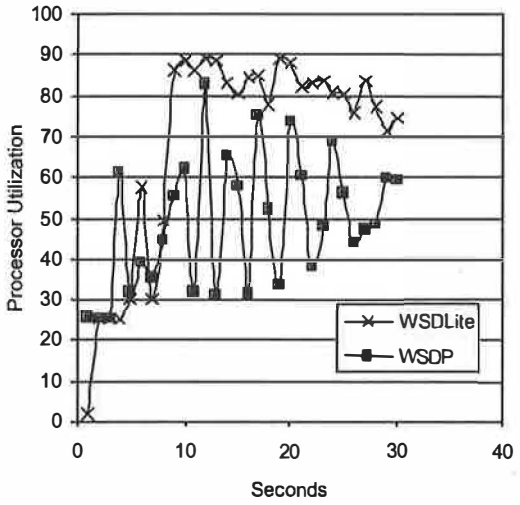


Figure 9. Processor Utilization for LU

Ilink and LU exhibit this effect to a larger degree because the computation-to-communication ratio of these applications is higher than that of FFT-3D or Barnes Hut. Therefore the contention for available processing resources is higher in these two applications, and the extra threads in WSDP exacerbate the problem. Figure 9 shows the average processor utilization of the four processors on one node when LU is run using WSDLite and WSDP. Data is shown for the first 30 seconds of the program's execution time, which represents the entire program execution under WSDLite. As indicated, the processor utilization for WSDLite remains high throughout the program's execution, resulting in a high parallel speedup for this application. When using WSDP, the processor utilization varies widely during the course of execution as the computation threads compete with the threads that implement WSDP. The resulting context switching reduces the effective processor utilization to the varying

levels shown in Figure 9, and results in a 60% increase in overall execution time.

6. Related Work

Previous work in this area can be divided roughly into two categories: new protocol and network interface designs, and attempts to deliver the improvements of these new network protocols and interfaces to applications. Our work falls into the second category, thus we concentrate on related work in this area.

Windows Sockets Direct Path (WSDP) [4] attempts to deliver the performance of user-level network interfaces (VIA in our case) transparently to TCP/IP networked applications written for the WinSock2 API. The approach taken by WSDLite differs from WSDP in two significant ways. First, WSDLite only implements a subset of the full WinSock2 API, albeit a useful subset that suffices for many networked applications. Second, our technique is not directly transparent. Although we do not require access to source code nor recompilation, we have to modify the applicable binaries using Detours in order for WinSock2 calls to be redirected. This is achieved by simply running the desired executable with a program called *withdll*, which injects the WSDLite DLL into the executable and rewrites the binary file to cause the redirect to the WSDLite implementation of WinSock2 functions.

VIA derives from a large body of related work in user-level communication, with the basic operation coming out of the U-Net research by von Eicken et al. [7]. As part of the U-Net research, a proof-of-concept implementation of TCP/IP was developed that delivered close to the raw performance of U-Net to TCP- and UDP-based applications. The results of this implementation, presented by the authors in [7], were partially what led us to investigate a performance alternative to the beta version of WSDP that we initially examined. VIA draws from several other research projects including application device channels [5], which provide the model for virtual interfaces to the network; and Virtual Memory Mapped Communication (VMMC) [6] and Active Messages (AM) [8], which provide the model for remote memory operations used in VIA. Other projects with similar goals to WSDLite and WSDP include Fast Sockets [11], which like WSDLite offers increased communication performance by collapsing protocol layers, using simple buffer management strategies, and by using "receive posting" to bypass data copying. Thekkath et al. proposed separating network control and data flow, and employed unused processor opcodes to implement remote memory operations [13]. Fast Messages [10] allow direct user-level access to the network interface, but do not support simultaneous use by multiple applications. The HP Hamlyn network implements user-level sends and receives in hardware [2]. ParaStation [14] provides unprotected user-level access to the network interface. With Active Messages [8], each message contains the address of a user-level handler that is executed upon message arrival with the message body as an argument. This allows the programmer and compiler to overlap communication and computation, thereby hiding latency.

7. Conclusions and Future Work

For those applications that use only the WSDLite subset of TCP functionality, we have demonstrated that WSDLite offers significant performance advantages relative to WSDP. However, this result must be qualified in several ways. First, we are using a beta implementation of WSDP. We expect the performance of subsequent versions of WSDP to improve. Second, some users may consider the modification of application binaries required by WSDLite in order to achieve transparency to be too aggressive for comfort. Third, while it is relatively easy to add additional functionality to WSDLite, certain aspects of Winsock2 functionality would likely be difficult to implement without incurring additionally overhead. In spite of these acknowledged limitations, WSDLite provides a useful tool for many applications.

We will continue to update our results as new versions of WSDP and the cLAN Winsock provider become available. We also intend to experiment with additional network applications. We are currently evaluating FTP and a web-based client/server database application for this purpose.

8. Acknowledgements

The authors would like to thank Galen Hunt of Microsoft Corporation for providing valuable information and sample code that allowed us to successfully use the Detours package with WSDLite.

This research was supported in part by grants and assistance from Compaq Computer Corporation, GigaNet Corporation, Intel Corporation, Microsoft Corporation, and by the Texas Advanced Technology Program under Grant No. 003604-022.

Bibliography

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, The NAS Parallel Benchmarks, NASA Ames RNR-91-002, August 1991.

[2] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pp. 245-259, 1996.

[3] Compaq Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture Specification, Version 1.0. 1997.

[4] Microsoft Corporation. *Windows Sockets Direct Path for System Area Networks*. Microsoft Corporation, 2000.

[5] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Annual Symposium on Operating System Principles*, pp. 189-202, 1993.

[6] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the International Parallel Processing Symposium*, pp. 388-396, 1997.

[7] T. V. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 40-53, December 1995.

[8] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 256-266, 1992.

[9] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[10] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, 1995.

[11] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the Usenix 1997 Conference*, January 1997.

[12] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, pp. 95-106, August 1997.

[13] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, October 1994.

[14] T. M. Warschko, J. M. Blum, and W. F. Tichy. The ParaPC/ParaStation Project: Efficient Parallel Computing by Clustering Workstations. University of Karlsruhe, Department of Informatics Technical Report 13/96, 1996.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.

Global Memory Management for a Multi Computer System

Dejan Milojicic, Steve Hoyle, Alan Messer, Albert Munoz, Lance Russell, Tom Wylegala, Vivekanand Vellanki,† and Stephen Childs‡

HP Labs, Georgia Tech,† and Cambridge University‡

[dejan, hoyle, messer, bmunoz, lrussell, wylegal]@hpl.hp.com vivek@cc.gatech.edu† Stephen.Childs@cl.cam.ac.uk‡

Abstract

In this paper, we discuss the design and implementation of fault-aware Global Memory Management (GMM) for a multi-kernel architecture. Scalability of today's systems is limited by SMP hardware, as well as by the underlying commodity operating systems (OS), such as Microsoft Windows or Linux. High availability is limited by insufficiently robust software and by hardware failures. Improving scalability and high availability are the main motivations for a multikernel architecture, and GMM plays a key role in achieving this. In our design, we extend the underlying OS with GMM supported by a set of software failure recovery modules in the form of device drivers. While the underlying OS manages the virtual address space and the local physical address space, the GMM module manages the global physical address space. We describe the GMM design, prototype implementation, and the use of GMM.

1 Introduction

GMM manages global memory in a Multi Computer System (MCS) by allowing portions of memory to be mapped into the virtual address spaces managed by each local OS. An MCS allows booting and running multiple operating systems on a single hardware architecture (see Figure 1) with cache coherent memory sharing among the nodes. Each node contributes its own physical memory divided in two parts. One is visible locally while the remainder contributes to the global memory, visible to all nodes (see Figure 2). The primary GMM benefits on a multi computer system are improved scalability and high availability. Scalability is improved beyond the scalability limits of a single OS, by allowing applications to run on any OS instance and some of them on multiple instances at a time, concurrently, while sharing memory and other global resources. Whereas the former require no modification, the latter require some amount of parallelizing and use of GMM and MCS interfaces.

Availability is improved compared to a large SMP by allowing other instances of local OSes, as well as applications on top of them, to continue running even if a single instance of an OS fails due to a software failure. Multi computer systems are especially well suited for enterprise data centers where applications, such as Oracle or SAP, require increased scalability and high availability.

GMM offers other benefits: first, the ability to use the fastest form of interconnect in an MCS system; second, the possibility of easy and fast sharing between nodes, following an SMP programming model; third, it allows for better resource utilization by allowing overloaded nodes to borrow memory from underutilized nodes; finally, it allows scaling of applications requiring memory beyond a single node (e.g. OLTP and data base).

GMM design goals consist of the following:

- scalability and high availability,
- shared memory within and among different nodes,
- a suitable environment for legacy applications designed to use shared or distributed memory,
- sophisticated support for new applications, and

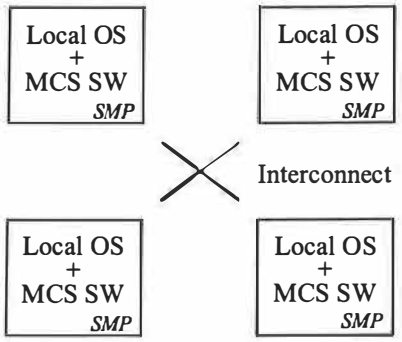


Figure 1 Multi Computer System (MCS) Architecture. *Each node (an SMP) runs a copy of the OS. Interconnect maintains cache coherent shared memory among nodes*

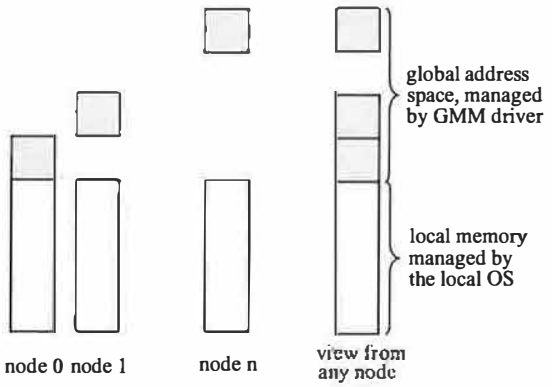


Figure 2 MCS Physical Address Space Organization. *Each node contributes a portion of its memory to global pool managed by GMM. Local memory is managed by the local OS.*

- good performance and resource sharing across nodes.

The rest of this paper is organized in the following manner. Section 2 presents GMM problems and non-problems. Section 3 presents the GMM design and Section 4 the prototype implementation. Section 5 overviews the use of GMM. In Section 6, we describe experiments. Section 7 presents the lessons learned. Section 8 overviews the related work. Section 9 concludes the paper and outlines the future work.

2 GMM Problems and Non-Problems

In the course of the GMM design we have identified problems that we considered important to address:

- **Recoverability.** Software failures on any node (e.g operating system failures) should not cause failure or rebooting of the whole system. GMM needs to recover its data from the failed node if it is still accessible. The memory occupied by a failed node needs to be freed up and inconsistent state needs to be made consistent (e.g. if a thread on a failed node died in the middle of updating data structures on another node). In order to be able to recover from the failure, the applications need to adhere to the recommended recoverable programming model (e.g. register for node crash events, replicate, checkpoint, etc.).
- **Memory scalability.** GMM is required to support access to more memory than is supported by a single OS, since there are *n* nodes each contributing to the physical memory of the whole MCS system. Therefore, limitation of 4GB of virtual address space size for ia32 is not acceptable [15, 16].
- **Local and remote sharing.** GMM must support memory sharing between threads on same and on different nodes, at both user (applications) and kernel (between MCS system components only) levels. This may require changes/extensions to the local OS APIs.
- **Usability and deployment.** The GMM recoverable programming model should not require significant changes to the existing applications. This is not such a strong requirement for MCS system components.
- **Minimal (if any) changes to the underlying operating system.** We used only extensions in the form of device drivers for prototype implementation. We also identified minimal changes to OS required for more sophisticated support (see Section 4.4 for details).
- **Globalization of resources and security.** Globally shared memory needs to be accessible for use from any node and it needs to be protected from misuse.

Based on experience from past systems and by adhering to Lampson's principles [18], we identified these problems that we decided to avoid solving:

- **Software distributed shared memory (DSM).** In MCS, consistency is supported by hardware. Other DSM systems supported recovery as a part of their consistency model (e.g. [5, 17]). In the case of GMM, recoverability is considered as a separate issue.
- **Local to remote memory latency ratio.** Our assumption is that the remote to local ratio will be 2 or 3 to 1 and as such it does not justify implications on the design. Early NUMA architectures had over 10-15 to 1 ratio and they paid a lot of attention to data locality. However, GMM recoverability still imposes some location-awareness, e.g. for replication purposes.
- **There is no single system image aspect.** GMM does not strive for single system image support, such as in the case of Locus [25] or OSF/1 AD [31], or for a transparent extension of the local interfaces, such as in the case of Mach [2]. It is acceptable to use GMM by writing according to specific GMM interfaces.
- **Transparent fault tolerance is not a goal.** GMM should be recoverable, but it is acceptable that certain users of GMM fail if they do not adhere to recoverable programming model. GMM guarantees to a recover from a single node OS failure (blue screen) which is the most common failure on NT. Gray claims that most failures are due to software [14]. In this phase, we have not addressed hardware faults.

3 Design

3.1 MCS Overview

GMM is designed to use the MCS system and recovery components. The recovery components implement a recovery framework to keep track of the current state of the nodes in the system. In the presence of failures, they detect and signal the faults to MCS components. The MCS system components provide support for global locking and fast communication (see Figure 3). Inter-Node Communication supports fast point-to-point, multicast, and broadcast communication of short messages, as an alternative communication model to shared memory. It is used for example for communication to nodes which may not yet have managed global memory. Also, it is a way of containing memory failures which is not possible if global memory is used.

The system knows how to recover global locks taken by failed nodes. Membership services provides support for the notion of a collective system. It relies on global

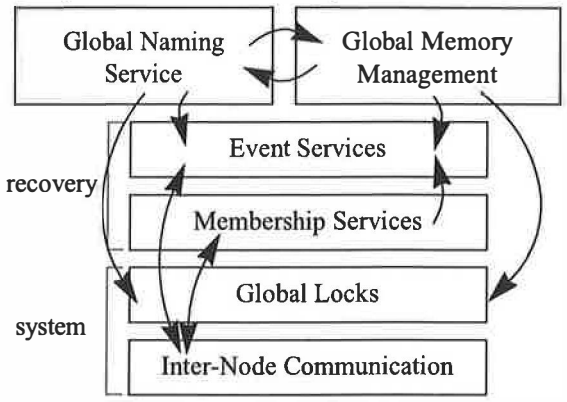


Figure 3 GMM and its Relationship to other Components *(recovery, GNS, global locks, etc.).*

locking and inter-node communication. It supports the software interface to actions which cause loss of membership, typically fault handling. The changes present in the system (both hardware and software) are reflected by global predicate-based Event Services. They notify registered components using callbacks of the type of event that has occurred. Each component is responsible for registering with Event Services its interest in important events. On each occurrence, each component is responsible for reacting to the event, which typically requires recovering the consistency of data structures.

In this way, the MCS system components act as an additional recovery service to allow aware applications and MCS system components to recover. To best ensure recovery, these components are written to expect failures in their operation. In order to further reduce the probability of failure of the recovery service itself, the complexity of these components is minimized.

Using this substrate, GMM implements management of the nodes' combined global shared memory. GMM references regions it allocates by unique identifiers, Global IDs (GIDs). The GIDs are obtained through the Global Name Service (GNS) which also uses the MCS system components. This Global Name Service provides the namespace for sharing objects in an MCS system.

GMM and other MCS components coexist with the host operating system as device drivers which use host OS services and provide a user API through a device driver and an access library in user-space (see Figure 4).

3.2 Physical Memory Management

Management of the global space is provided by GMM running on each node in the system. In order to communicate, these instances share data structures in global memory, protected by global locks. The root of the data structures is the Master Table which maps from GIDs to

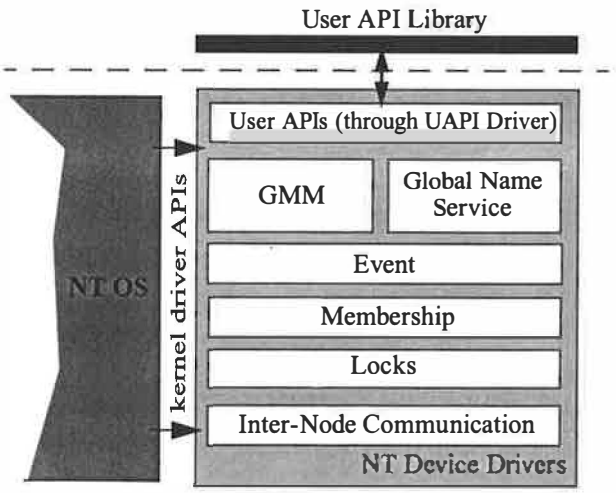


Figure 4 GMM Organization and APIs. *GMM is implemented as a device driver, using the kernel driver interfaces. It exports user and internal APIs for the other MCS components.*

particular allocations of memory, called Sections. Each node has space for a Master Table, but only two copies are used at a time (primary and replica).

Each node then has Section, Sharer and Free memory tables which describe the allocations from global memory in its managed portion of global memory. The Section table describes each region of memory allocated and indicates nodes sharing that Section. The Sharer table then describes which processes (from which nodes) are using memory on this node. Finally, the Free memory list is the usual data structure to hold unallocated memory managed by this node (see Figure 5).

By maintaining two copies of the master table (which are updated on each access) the GMM data structures can always be found upon a single node failure. Other enhancements to these data structures have also been made to ensure the data in the tables can be recovered on a failure (see Section 3.4).

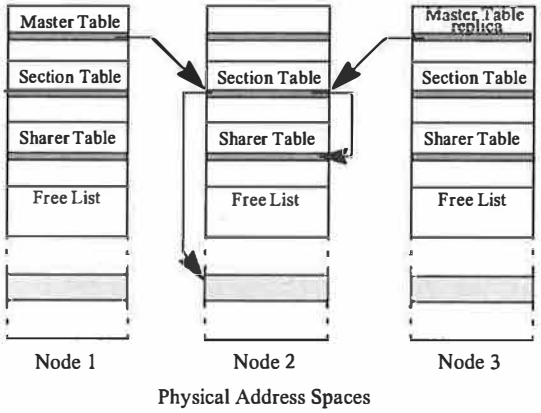


Figure 5 Physical memory management data structures in MCS consist of *the master table (unique for MCS, but replicated), section table, sharer, and free list (per node).*

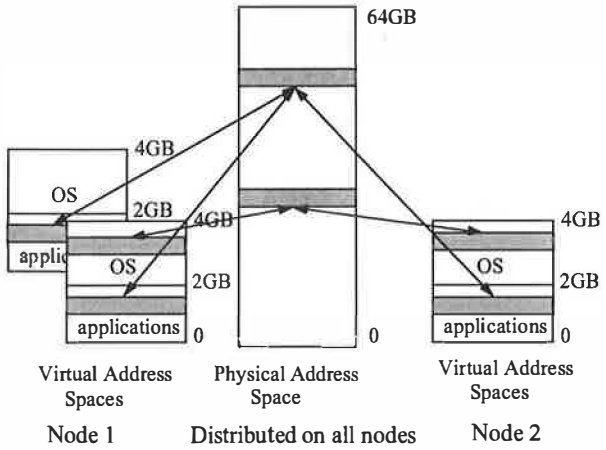


Figure 6 Memory Sharing in MCS: *Applications or MCS kernel components may map the physical global memory into their virtual address spaces and share it with other nodes.*

3.3 Sharing

Applications and MCS system components use GMM to share regions of global memory among themselves. GIDs, obtained from the Global Name Service, provide the naming for these sections. Once identified, these GIDs are passed to GMM either to allocate or share an existing region of global memory. If allocation is requested, then physical memory is reserved using the above structures and mapped into the caller's virtual address space. If a caller opens an existing GID, the desired region is simply mapped into its address space (see Figure 6).

Mapping cannot be performed by using direct control over the virtual memory hardware since the host OS is already using this hardware. For coexistence with the host operating system, it is necessary to be able to use the host OS to map the physical memory into the address space of a particular process. This has the advantage of providing automatic memory protection on a per process basis as long as the host OS supports it.

In the current prototype, global memory is not paged due to limitations of the NT kernel. However, with the ability to add an external page fault handler to the operating system for the global memory, a page system could be implemented. Such a paging system could then take advantage of vacant memory on other nodes to improve performance as a fast backing store before placing copies on stable storage for reliability [11].

3.4 Recovery

The recoverability objective of the GMM design is that surviving nodes be able to recover from the failure of any single operating system on any node in the system. Such failures are detected by a software heartbeat per node monitored by other instances of the membership service in a ring. The failure is signaled using the Event Service to any interested system component. This event causes GMM on each node to recover its data structures to a consistent state by performing the following tasks:

- for remote memory used by the failed node, the failed node is removed from the sharer list of this memory,
- for remote memory that is being allocated/released by the failed node, the operation is completed or aborted,
- if the failed node contained master or replica, a new master/replica is allocated on a surviving node, and
- the free list is updated, if memory is no longer shared.

If the failure also caused loss of the node's resources, e.g. shared memory, then a separate event is issued which causes GMM to:

- remove access to those lost sections and bring its data structures into consistency.
- flush the node's caches.

The biggest recovery challenge comes when an OS crashes during memory allocation. This could result in some of the GMM tables being partially updated or locked but not released. As an example, consider a failure when the node allocating memory fails after identifying a portion of physical memory for use. Since this portion of physical memory is no longer present in the free list, this portion of memory would be unavailable for allocation and hence would be lost.

To overcome this problem, global memory allocation is implemented as a two-phase nested transaction. In the first phase, all the required resources are reserved. After reserving, the necessary data structures are updated. Finally, these resources are acquired. In the case of an inability to acquire a particular resource, the reserved resources are released. During global memory allocation, the following resources are needed:

- A Master Table entry to map the GID associated with this portion of global memory to the identity of the node and the section of the physical memory. This is maintained on the primary and replica for reliability.
- A new descriptor in the Section Table on the node hosting the physical memory to map the section with the physical address. The section descriptor maintains an entry into the sharer descriptor table.
- A new descriptor in the Sharer Table to include the identity of the caller node in the sharer list.
- Physical memory from the Free List to associate as this portion of global memory.

By implementing a two-phase transaction system, new entries to each table are marked as temporary until all the resources can be acquired and all the updates can be made. Only then are the new entries in each table committed. During recovery, all resources that had been reserved by the failed node and not committed are reclaimed. Recovery completes only after all uncommitted resources reserved by the failed node are reclaimed.

Once system components have recovered, applications too can respond to system failure events signaled by the recovery components and use this to increase the availability of their service. Of course, application failures themselves are still the responsibility of the application to detect and handle itself.

3.5 Locking

To ensure consistency using locking, especially in the context of node failures and recovery, the MCS platform was designed to support hardware spinlock primitives. This design allowed the lock subsystem to be completely recoverable from single node or memory subsystem failures. If a node hosting a spinlock failed, it was designed to redirect (in hardware) the lock to a new node and was set to return an error code of further access. Clients using the locks and receiving this error code, would cooperate to recover the state of the lock.

Consider the case that surviving GMM code was holding a lock at the time of the crash. In this case, when exiting the critical section it would receive an error code and could reset the lock state and recover synchronization. If, however, GMM code execution was lost with the lock held, further acquisition would be prevented from accessing the critical section until the recovery code recovered the data structure and reclaimed the lock by resetting it.

Unfortunately no implementation of this hardware locking support was available on any existing platform. So an emulation of the desired semantics was implemented using NT's spinlocks. This emulation provides for error code reporting when the locks are lost and allows lock resetting by the recovery process.

3.6 I/O

I/O may occur during single node or memory failure. Local I/O operations have the same recovery semantics as a traditional NT system, but problems arise when I/O is made to remote systems through global memory. Existing I/O operations to failed memory are taken care by the platform hardware. While client access to the data requires careful use of locking for mutual exclusion and checking for device error codes to avoid consuming erroneous data.

3.7 Application Recovery

Application recovery is the sole responsibility of the application. Applications choosing not to support any form of recovery can either continue regardless (hoping no ill effects will result), respond to signals by the system, or leave restart/recovery to a third party (such as an application monitor).

Applications are signalled a failure event either before or after the system has fully recovered depending on the type of event. If an OS crash caused data inconsistency or hardware was lost, the system recovers first and then signals the application to make itself consistent. However, if resources are manually removed (e.g. shutting down a node for an upgrade) then, after initially informing the system components, the application is signalled first. It is then allowed some time to recover before the system components fully recover from the lost resources. This allows the application to copy existing data and reposition resources before the system attempts to forcibly revoke lost resource allocations. It also improves performance of recovery by eliminating unnecessary recovery of resources the application will release itself.

4 Prototype Implementation

In order to experiment with our approach, a prototype implementation has been created under Windows NT 4.0 Enterprise Edition. The underlying machine is a 4 node multi computer with an SCI interconnect providing hardware coherent shared memory. Each node is a 4-way Pentium II 200Mhz machine with a Gigabyte of memory contributing 256Mb to local memory for the host OS and 768Mb to the global memory pool. NT is informed using the `/maxmem` command in the NT Loader to manage only the lower 256MB region.

4.1 Integration with NT

GMM and other MCS components coexist with the host operating system, running in kernel-mode implemented using the NT Device Driver Kit (DDK) [30].

Each MCS component is written as a separate device driver to provide a modular system design, each exporting an internal API to the other components. A control device driver synchronizes the initialization of the device drivers and the MCS system software.

At boot time, the MCS software on a single node (the primary, defined as the alive node with the lowest *id*)

coordinates the boot, allowing each additional node to enter the system one at a time. As each node joins, this event is communicated to other nodes using the fast inter-node communication. This allows the system to build the membership of the system and communicate any events to members. Once these are initialized, the global memory manager initializes. GMM and other MCS components are designed to allow rejoin of failed nodes, even though the current development platform does not allow for this (limitation of the firmware).

GMM starts by initializing empty versions of its tables and then it contacts the current primary node in order to reference the master tables. At this time, GMM has registered itself for OS failures with Event Services. If this node discovers as it joins the system that there is no replica, it copies the master table and informs all other current nodes that it wishes to be the replica. Only one node enters at a time, so there are no race conditions.

Although various MCS components (GMM, GNS, etc.) are implemented as kernel device drivers, they actually function as shared library components in the kernel. A separate MCS component, called the UAPI driver, is registered with the operating system as a driver. It receives requests from user space in the form of *ioctls* (I/O controls), which it translates into procedure calls to the appropriate MCS components. An MCS DLL (Dynamic Link Library) manages all of the *ioctl* communication with the driver, presenting user space applications with an explicit procedural interface.

The UAPI driver also provides generic MCS bookkeeping services. It keeps track of all user space processes that call MCS APIs, notifying MCS components when a process exits so that accurate reference counts can be kept, data structures can be properly cleaned up, and system resources can be recycled and reused.

The UAPI driver also maintains process indexed mappings for all MCS kernel objects (global shared memory segments, global mutexes, and global events) created or opened by any user process. This relieves the individual MCS components from validating and translating the GIDs. For example, a user process provides a GID when it calls the API provided to map a global memory segment into its address space. In responding to this call, MCS must first find a corresponding data record and verify that the given process has access to the corresponding memory segment. Both of these tasks are done by the UAPI driver before calling GMM.

4.2 GMM APIs

The primary goal of the GMM user space APIs is to provide a convenient interface allowing processes on different nodes to access the same memory resources. Included is a strong foundation for different processes to use and maintain identical virtual address mappings to shared memory, regardless of where the processes run.

Maintaining the same virtual address mappings to shared memory is significant because it allows applications to use direct memory references to or within shared data structures. This in turn allows the virtual address for any memory location to serve as an object identifier as well as a memory access handle. A linked list with the links implemented as direct memory references is a typical example. Using virtual addresses in a dual role (for identifiers as well as access handles) is prevalent in Windows NT programming.

Applications need to be organized as multiple processes to take maximum advantage of the availability and recoverability features. The MCS system software is designed to confine the effect of OS failures to a single node. Recoverable applications are expected to do the same. When a node goes down, there must be application processes already running on other nodes in order to recover.

Being able to maintain virtual address mappings to shared data structures through recovery operations in response to faults is an important part of the support provided. Consider a recoverable application maintaining two copies of global shared data in such a way that at any given point in time, one copy or the other is always in a consistent state. Typically, there will be a primary copy that is directly accessed during normal operations, and a secondary copy that is updated only on transaction boundaries of coarser granularity.

When a failure occurs that compromises the primary copy, application recovery uses the secondary to restart computations from a consistent point. This is most quickly facilitated by promoting the secondary to become the primary. However, it requires a change in the virtual to physical memory map for each of the application processes, or that recovery by the application includes repairing all of its memory references to the primary. The latter would be particularly error prone, even for highly disciplined programmers. Moreover, it would pose a significant obstacle for attempts to modularize recovery code from normal operation code. Thus, the GMM user space APIs allow processes to reserve virtual address ranges, which can be freely mapped and remapped to different sections of physical

memory. The GMM APIs can be classified into the following groups:

1. Reserving and unreserving virtual address ranges.
2. Acquiring and relinquishing access to identified physical memory resources.
3. Mapping and unmapping specific virtual address ranges to specific physical memory resources.

Keeping these groups independent of each other as much as possible is the key for allowing multiple processes to maintain the same virtual mappings to shared memory. After reserving a given range of virtual addresses and acquiring access to a given segment of physical memory, a specified portion of the virtual space is mapped to a specified portion of the physical memory. This mapping can be undone without losing reservation and portions of it can be remapped to other segments of physical memory.

4.3 Security

Security in MCS builds on the access control mechanisms provided by NT. Each object secured by NT has an associated security descriptor which contains an access control list (ACL). When a user attempts to access an object, the security descriptor is consulted and the access is verified against the ACL. MCS must maintain a globally valid association between each MCS kernel object and its security descriptor. This allows security information to be retrieved even when a user accesses an object (e.g. a shared memory segment) located on another node. MCS subsystems that provide global objects must be modified to perform security checks using this information.

There are two issues involved in managing globally accessible security descriptors: storage and lookup. The security driver stores the descriptors in a table in global shared memory. This table is identified by a GID, and the combination of this GID and an offset within the table make up a globally valid address. This address is then stored with the GID for the protected MCS object, making it possible to retrieve the security descriptor when the object is accessed.

MCS security is implemented by a security driver and some modifications to the subsystems that provide kernel objects. The security driver implements routines to assign a security descriptor to an object and to retrieve the security descriptor for a particular object. Other subsystems are modified to use these routines when creating new objects and verifying accesses.

4.4 Issues with Extending NT for GMM

During our implementation, we encountered three problems with integrating our system with the NT kernel.

Reservation of the virtual address space. Since GMM uses shared data structures one of the most convenient ways to implement the data structures is using pointers. In order to use this optimization we need to have the same virtual memory address across all nodes of the system. Unfortunately, under the NT kernel, there is no way to guarantee the virtual address allocated to the global memory mappings. Instead it is only possible to create mappings as early as possible to hopefully receive the same address. Rather than relying on this ad-hoc solution, the data structures instead are implemented through table indexing. While this is not a great problem, it does reduce the readability of the code base.

Intercepting page faults. Our second problem limited our implementation and consideration of adding paging to our system. It appears that there is no way in the NT kernel to add an external pager for a region of memory. Since our system exists along side NT rather than inside it, Windows NT does not manage the physical memory of the global pool and so we would need to add our own separate pager in order to manage this address space.

Scaling memory beyond 4GB. Windows 2000 supports Address Windowing Extensions (AWE) interfaces for using more than 4GB of physical address space. AWE allows multiple processes to use more than 4GB of physical address space. In addition, a single process can use more than 4GB in a limited way (only 4GB can be mapped at a time, since virtual address space is still limited to 4GB). The AWE interfaces represent a step in the right direction, however, they fall short of the GMM requirements with the inability:

- specify the physical addresses to be mapped to a certain virtual address space: the AWE returns free (non-contiguous) physical pages,
- reserve physical address space for GMM, i.e. NT should not allocate the physical ranges shared between nodes to other local mappings, and
- separate *unmap* from *free*: AWE supports *reserve* virtual address space, *allocate*, *map*, and *free* physical pages; in the absence of inter-node sharing, there is no need for *unmap*, it is achieved as a part of *free*.

New 64-bit processors (e.g. ia64) will relieve some of the problems encountered with designing and implementing GMM. First, the 4GB limitation would go away. Second, because of the large virtual address space, sharing the space among processes would be eas-

ier to implement (space could be reserved ahead for sharing purposes). Next, the recovery model will be improved, especially the memory failures.

4.5 Limitations of the GMM Prototype

The current GMM prototype implementation under NT has the following limitations:

Premapped and non-paged global memory. All global memory is pre-allocated in the NT non-paged system virtual address space and subsequently allocated from this pool, since that is the only way NT will permit dynamic mappings of the memory into the user portion of a process' address space.

Incompatibility with the local OS semantics. Examples include address space inheritance, security, etc. This limitation is introduced because the local OS is not managing GMM memory and it is not in the position to handle it in accordance with the GMM requirements. In order to make this possible, the GMM interfaces need to be used to achieve security, sharing, recovery, etc.

Inability to test memory failures. Given the reliance of the current hardware platform on an SCI ring, it is difficult to test for memory failures as a result of an entire node failure rather than a simple OS crash. A node failing cannot be simulated by breaking the SCI ring to isolate a node without disrupting communication to all nodes. Instead, we simulated memory failures by explicitly unmapping memory which would in normal operation be mapped as part of global memory.

5 Using GMM

5.1 Shared Memory Programming Models

An MCS global application is a set of one or more cooperating processes that run on the nodes of an MCS system. There may be multiple processes per node, and the processes may be multi-threaded. MCS global applications that have been modified to take advantage of an MCS system use the global memory to achieve two benefits: performance scalability and high availability. By performance scalability we mean that the throughput of an application should increase in proportion to the amount of computing resources allocated to it. For example, an MCS application which has its processes running on two nodes of an MCS system should deliver roughly double the throughput of an application running on one single node. In this context, high availability means that the application can continue to provide service to users in the event of OS (and in future hardware) failures on any node in the system.

An application must have certain characteristics to be able to exhibit performance scalability while using GMM. The first characteristic is the same as in the case of an SMP system: the application must consist of independent threads, and the throughput of the application must increase with the number of concurrent threads. The shared data set can be placed in global shared memory where it can be accessed by all processes comprising the global application. There are two keys to achieving high availability. First, each process must maintain its state in global shared memory, so that if a process terminates (for example, due to the crash of the operating system on its node) then the task it was executing can be completed by another process on another node. Second, each data item stored in global memory should be backed up by the application to a redundant copy, either in global memory residing on a different node or on disk. If memory is lost at a result of a node failure then the redundant copy can be referenced.

5.2 Kernel Components that Use GMM

MCS kernel components can use global shared memory to their advantage. In Windows NT (and other OSes), the kernel and privileged-mode device drivers share a common virtual address space. While the Windows NT kernel itself does not use global shared memory, it permits privileged-mode drivers to map global memory into the kernel address space. Although all kernel components can access all mapped global sections, the common practice is for MCS kernel components to share sections only with their counterparts on other nodes.

GMM uses its own services: the tables describing the global memory sections attributes and the locations of unallocated ranges of the global memory are themselves stored in global shared memory. This permits distributed management of the global memory resource. GMM on any node can make a new global memory allocation by updating the shared tables.

Global networking relies on global shared memory as its physical transport medium. This component presents itself to the operating system as a standard networking driver, so all standard networking protocols are supported. To send a packet from one MCS node to another, the sender places the packet in a buffer in a global memory section and posts an interrupt to the receiving node. The global memory sections required to hold the buffers are allocated when nodes initialize their networking. Each node creates a global memory section of about 2 MB on each other node to which it can send packets. For each buffer, there is only one node that writes and only one node that reads the contents.

Another MCS kernel component implements global synchronization objects, mutexes and events, for use by user space applications. These objects have the same semantics as the Win32 mutexes and events, except that they can be used from any process on any node. These objects also have recoverability features allowing them to survive node and resource failures in the system.

The final example of a kernel component using global shared memory is the global file system. This file system presents a common global file tree across all MCS nodes. The file system maintains a cache of recently used file blocks in global shared memory, so that performance is increased for accesses to shared files (opened by multiple processes simultaneously). The implementation of the global file cache has not been completed, so performance statistics are not available.

5.3 Applications that Use GMM

An application whose characteristics should benefit from the properties of global memory management is a database manager. Most database managers can increase their throughput in proportion to the number of concurrent threads accessing (different parts of) the data set. In current practice, it is common to have data sets many gigabytes in size, and many workloads feature a significant number of update operations. We considered porting a major database manager, such as Oracle or Informix, to MCS, but the size and complexity of the code base would make this a difficult undertaking.

The first global application that is ported to our prototype MCS system is a main-memory database manager from TimesTen Performance Software. This product supports the same query and transactional update functionality of a conventional database manager, but with increased performance and predictability of response time owing to the fact that the entire data store resides in main memory. The structure of the application makes it simple to place the data store in global shared memory and to synchronize access to it from multiple nodes.

The second MCS global application is a Web server modified to share Web caches in global memory. This application was chosen to allow future study of availability and performance issues in potential future Internet systems. Both the Microsoft IIS and Apache Web servers were modified to manage a cache of recently-used files, rather than relying on the file system's cache. The cache is placed in global shared memory so that each node could read cached files and copy new files from the file system into the cache. A Least-Recently Used (LRU) global cache management algorithm was used. Results in this paper represent initial demonstration figures of this work using the Microsoft IIS Web server only.

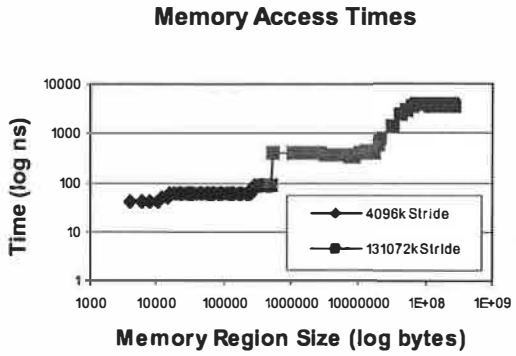


Figure 7 Memory Access Times on the Prototype: *Two memory region tests cover the entire memory hierarchy.*

6 Experiments

In order to obtain the baseline performance of the prototype configuration, we used lmbench [20] to measure the latency of the various levels of the memory hierarchy. Lmbench calculates the minimal memory access time by traversing the memory region at various step distances (strides) to determine cache effects. By using two memory region size ranges and two strides to best determine cache effects (4Kb-512Kb with access stride 4096 bytes and 512Kb-256Mb with access stride 128k) we were able to effectively evaluate the complete memory hierarchy access latency, including first and second level caches as well as local and remote memory. This is achieved by using memory list structure those elements are placed at a certain stride through a memory section (larger than the cache size) from local and shared memory. In our experiment, we measured the latency for the level 1 cache latency to be 40*ns*, the level 2 cache latency to be 50~85*ns*, non-cached local memory (both local DRAM and the local SCI cache) to be about 304~314*ns*, and the remote memory load access latency to be approximately 3950-4125*ns* (see Figure 7).

Another set of performance measurements compares the use of networking over shared memory v. loopback. The results are presented in the table below. Shared memory networking demonstrates relatively good performance since large blocks of data are being transferred.

Measurement v. Environment	Loopback *(Kb/sec)*	Shared Memory Network *(Kb/sec)*
FTP Transfer: 30MB (binary)	*9286*	*7870*
TTCP Transfer: 2K writes of 8KB	*10233*	*8197*

We measured the performance of the MCS version of the TimesTen database manager by using a debit/credit benchmark patterned after TPC-B. Our data set had

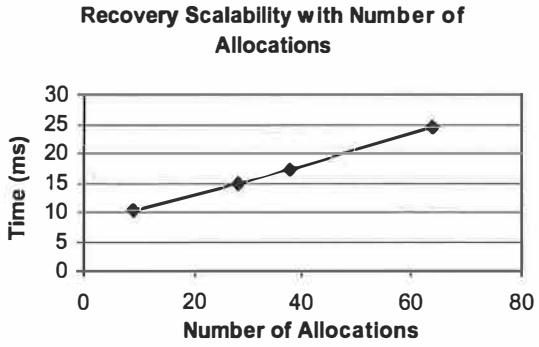


Figure 8 GMM Recovery Scalability: *Recovery was timed on the four nodes when one failed with an increasing quantity of shared memory allocations.*

90,000 rows. Our first experiment compared the released version of TimesTen with the MCS version on a uniprocessor with the data set stored in memory local to the node. This showed that the overhead of replacing native Windows NT functions for memory management and process synchronization with the corresponding MCS functions was approximately 1%. Our second experiment compared the MCS version on a uniprocessor with the data set stored in memory local to the node with a similar configuration with the data set stored in remote memory. The two runs showed virtually identical performance, showing that our data set fit in the system's remote memory cache of 32 MB, whose response time was the same as for local memory. Finally, we compared the released version on a four-way SMP with four MCS nodes, and again the performance was similar, because the execution time was dominated by contention for the lock on the data set.

Experiments on the recoverability of the GMM modules on three nodes in presence of the failure of the fourth node were performed. These experiments measured recovery time of GMM modules in the MCS system and scalability with number of allocations of GMM recovery. In our experiments we fail one node (blue screen it) and the other three nodes successfully recover. Our experiments showed that the time to recover GMM takes approximately 10.2*ms* for one shared application memory region (plus 8 taken by the MCS system components). This represents a significant improvement in service disruption time compared to the time to reboot the sharing nodes. Experiments also showed that as these were scaled to a reasonable maximum of 64 shared allocations, the overhead to recover scaled linearly (see Figure 8). These figures include the time to signal the failure, recover on all remaining nodes and resynchronize on completion.

Initial experiments with a Web Server application were performed to demonstrate the use of the GMM modules in a real world application. Using two nodes of our prototype, we ran the Microsoft IIS Web server with a modified shared memory cache. Then a popular web benchmark was run to provide a workload benchmark for these two web server machines. Four client machines were used to generate sufficient workload to maximize the work of the servers, with the total workload spread across the two server machines. Each client ran four load-generating processes. Results were measured for just one machine with a 200Mb Web cache and two machines each contributing 100Mb to a shared GMM cache. The second configuration therefore has the same total cache size as the first configuration, but with the use of shared memory between machines has allowed the application to be scaled to two machines while maintaining any sharing. Results are presented in Figure 9.

Our experiments measured that the performance approximately doubled when processing and I/O resources were doubled. Such speedups are common with web servers since content can be replicated, but this requires doubling the memory resources also. In this experiment, the servers seem to scale with server resources while memory resources remained constant due to the sharing of resources. These figures, while interesting, are not designed to demonstrate specific performance benefits, since they are relative and entirely unoptimized. They do, however, demonstrate that the MCS system and in particular the GMM modules are capable of running real world applications and provide suitable potential for the real world use of such a platform. Once this work has been completed, we hope to publish further results.

7 Lessons Learned

1. It is possible to extend NT with global memory management without changes to the existing code base. However, this is only true for limited implementations, where memory is preallocated and pinned.

2. In order to achieve fully functional GMM there is a need for extensions to Windows NT (see Section 4.4

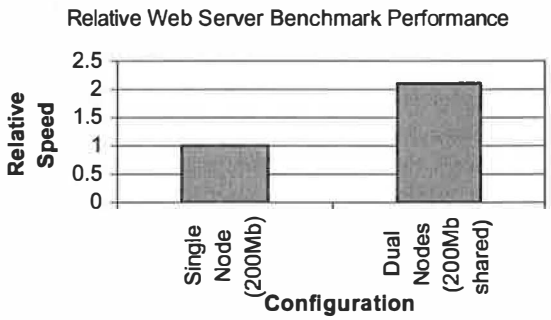


Figure 9 Web Server Benchmark Performance: *Throughput in single- and two-node shared memory configurations.*

for more details). If NUMA-like machines become widely spread, this effort is likely to be standardized among OS and hardware vendors.

3. It is hard to develop a flexible and easily deployable programming model acceptable for legacy applications. Applications need to be parallelized in order to allow for easy deployment using GMM. Kernel components are more likely to use GMM since their use is hidden from the users. An example is networking (see Section 5.2).

4. Recoverable programming models for GMM are even harder. They require careful design and replication of data structures. Furthermore, they are error-prone since recovery from partial updates can be complex. Memory failures (not addressed in the initial implementation, but considered long term) make this problem even harder and more hardware dependent.

5. Recovery poses a requirement to the local OS "not to get in the way". For recovery purposes, it is easier to provide extra code that pays attention to preserving the state, rather than relying on the existing (nonrecoverable) OS which may lose the state on the OS execution stack. This may duplicate functionality, but the same code can be used for different OSes.

6. Hardware support for containment and recovery is very important. We realized that in order to support recovery from hardware failures we need additional hardware support in order to notify and mask memory failures, which would normally cause the system to machine check and fail.

8 Related Work

There are a number of systems related to GMM both in academia and in industry. In academia, most related to our work are OSes developed at Stanford: Hive [6], DISCO [4], and Cellular DISCO [13]. For Hive, Teodosiu explored the possibilities hardware fault containment in multiprocessor systems [28]. Cellular DISCO is derived from the DISCO virtual machine work. DISCO showed that the scalability limitations of multi-processor systems could be overcome by running multiple virtualized OSes rather than scaling one OS. Cellular DISCO takes this virtualization further by using the ability to kill and restart virtualized OSes for fault isolation and containment. This approach does not consider recovery other than rebooting virtualized OSes. Our approach attempts to provide the ability to recover from software and hardware failures.

There has been a lot of work on memory management for early NUMA systems [3, 9], as well as for NORMA (distributed memory) model [1, 2, 12, 31], but none addressed failures. The cooperative caching project at the University of Washington investigated the use of memory from underutilized workstations [11]. Some of distributed shared memory systems address fault recovery [5, 17]. The Rio system addresses recoverability of the OS from SW failures, including wild writes [7].

Many companies provide 'high availability' systems through partitioning, such as Sun's UE10000 [27], Unisys's Cellular Multiprocessing Architecture [29], Sequent's NUMA-Q servers [26], Compaq's Wildfire/ OpenVMS Galaxy platform [8, 10], SGI's Cellular IRIX /SGI 2000 family [19] and IBM's S/390 Parallel Sysplexes [21, 24]. These systems provide increased availability by hardware partitioning, redundancy, and by running in "lock-step" (IBM's Sysplexes). These systems rely on hardware features to allow failures to be contained per partition (a logical node or set of nodes). By having such partitions and executing multiple OS instances, they provide the ability to contain the effects of software failures while since allowing shared-memory between instances for fast communication. On such systems, high availability software provides error reporting/logging and control of partitioning where applicable. But non-redundant software and hardware failures cause failure of particular partitions and are resolved by rebooting. Our work tries to increase the availability envelope by using software which can attempt to recover from dependencies on software crashes rather than requiring dependent partitions to reboot. This form of recovery is of increased importance when resources, such as memory, are being shared and partitions are therefore more tightly coupled. This type of sharing, as typified by common applications, such as Oracle's Database Server, is key to obtaining good performance [23].

9 Summary and Future Work

We designed and implemented a prototype implementation of global memory management for the NT OS. We achieved this without modifications to the OS. However, the prototype implementation has limitations, such as non-paged global memory. In order to remove these limitations, some modifications to the underlying OS are required. We described the required functionality missing in the existing NT implementation for fully functional GMM. We described GMM recovery as well as some applications that we used with it. Finally, we derived lessons learned.

Our future work will address memory hardware failures and specifically how to recover from them in case of the ia64 architecture. We believe that most of the failures that will remain will be due to software [14]. However, with the increased high availability requirements of scalable systems, the mean-time between failure of memory and other components, such as interconnections, processors, will increase. This may not be sufficient for systems such as enterprise data servers. Therefore we need recoverable programming models to fill the gap. In particular, we are interested in the tradeoffs between hardware and software support for optimal recoverable programming models.

Acknowledgments

Tung Nguyen is the father of the MCS program. Lu Xu implemented GNS and parts of GMM. Todd Poynor and Guangrui Fu conducted Web server and GMM latency experiments. Chen Li ported TimesTen to MCS. Jork Löser evaluated GMM on Linux. We thank TimesTen for letting us use their product. We are indebted to Herman Härtig, Keith Moore, Todd Poynor, and Mike Traynor for reviewing the paper. Their comments improved contents and presentation.

References

[1] Abrosimov, V., Armand, F., Ortega, M., I., "A Distributed Consistency Server for the CHORUS System", *Proc. of the USENIX SEDMS*, March 1992, Newport Beach, pp 129-148.

[2] Black, D., Milojicic, D., Dean, R., Dominijanni, M., Sears, S., Langerman, A., "Distributed Memory Management". *Software Practice and Experience,* 28(9):1011-1031, July 1998.

[3] Bolosky, W., Fitzgerald, R. P., Scott, M. L., "Simple but Effective Techniques for NUMA Memory Management," *Proc. 12th SOSP*, pp. 19–31, Wigwam Litchfield Park, Az, December 1989.

[4] Bugnion, E., Devine, S., Rosenblum, M, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *Proc. of the 16th SOSP*, Saint Malo, France, pp. 143-156, Oct. 1997.

[5] Cabillic, G., Muller, G., Puaut, I., "The Performance of Consistent Checkpointing in Distributed Shared Memory Systems", *Proc. of the 14th Symposium on Reliable Distributed Systems*, Bad Neunahr, Germany, September 1995.

[6] Chapin, J., et al., "Hive: Fault Containment for Shared-Memory Multiprocessors," *Proc. of the 15th SOSP,* pp. 12-25, Dec. 1995.

[7] Chen, P.M., et al., "The Rio File Cache: Surviving Operating System Crashes", *Proc. of the 7th ASPLOS, October 1996.*

[8] Compaq, "OpenVMS Alpha Galaxy Guide", Downloaded January 2000, http://www.openvms.digital.com:8000/72final/6512/6512pro.pdf.

[9] Cox, A., Fowler, R., "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum," *Proc. Twelfth SOSP*, pp. 32–44, December 1989.

[10] Digital Readies 500mhz-based 'Wildfire' Server", PC Week, March 1997.

[11] Feeley, M.J., et al., "Implementing Global Memory Management in a Workstation Cluster", *Proc. of the 15th SOSP*, Dec. 1995.

[12] Forin, A., Barrera, J., Sanzi, R., "The Shared Memory Server", *Proc. of the Winter USENIX Conf.*, San Diego, 1989, pp 229-243.

[13] Govil, K., et al., "Cellular DISCO: Resource management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. of the 17th SOSP,* pp. 154-169, December 1999.

[14] Gray, J., and Reuter, A., "Transaction processing: Concepts and Techniques," Morgan Kaufmann, 1993.

[15] Intel, "Intel Architecture Software Developer's Manual", and "Addendum", 1997.

[16] Intel, "The Intel Extended Server Memory Architecture", 1998.

[17] Kermarrec, A-M., Cabillic, G., Gefflaut, A., Morin, C., Puaut, I., "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", *Proc. of the 25th Int'l Symposium on Fault-Tolerant Computing Systems*, pp 289-298, June. 1995.

[18] Lampson, B., "Hints for Computer System Design", *Proc. of the 9th SOSP*, October 1983, pp 33-48.

[19] Laudon, J., Lenoski, D., "The SGI Origin: A ccNUMA Highly Scalable Server", *Proceedings of the 24th International Symposium on Computer Architecture*, pp 241-251 June 1997.

[20] McVoy, L., Staelin, C., "lmbench: Portable Tools for Performance Analysis", Proc. of USENIX 1996 Conference, San Diego, CA, January 22-26, 1996, pp 279-294

[21] Nick, J.M., et al., "S/390 Cluster Technology: Parallel Sysplex", *IBM Systems Journal,* v 36, n 2., 1997.

[22] Oracle, "Oracle Parallel Server in the Digital Environment, High Availability and Scalable Performance for Loosely Coupled Systems," Oracle White Paper, June 1994.

[23] Oracle 8 Support for the Intel^{R} "Extended Server Memory Architecture: Achieving Breakthrough Performance," *Intel Document, Oracle's Note,* 1998.

[24] Pfister, G., "In Search of Clusters", Prentice Hall, 1998.

[25] Popek, G., Walker, B., "The Locus Distributed System Architecture", *MIT Press Cambridge Massachusetts,* 1985.

[26] Sequent White Paper, "Application Region Manager", Downloaded in January 2000 from http://www.sequent.com/dcsolutions/agile.pdf.

[27] Sun Microsystems, "Sun Enterprise^{TM} 10000 Server: Dynamic System Domains", White Paper, Downloaded January 2000, http://www.sun.com/datacenter/docs/domainswp.pdf.

[28] Teodosiu, D., et al., "Hardware Fault Containment in Scalable Shared-Memory Multiprocessors," *Proc. of the 24th ISCA*, June 1997.

[29] Unisys, "Cellular MultiProcessing Architecture," White Paper, Downloaded January 2000, http://www.unisys.com/marketplace/ent/downloads/cmparch.pdf.

[30] Viscarola P. and Mason, W.A., "Windows NT Device Driver Development," *Macmillan technical Publishing,* 1999.

[31] Zajcew, R., *et al.*, "An OSF/1 UNIX for Massively Parallel Multicomputers", *Proc. of the Winter USENIX Conference,* January 1993, pp 449-468.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Earthlink Network
Greenberg News Networks/MedCast Networks
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Sendmail, Inc.
Server/Workstation Expert
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
UUNET Technologies, Inc.
Web Publishing, Inc.

Supporting Members of SAGE:

Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org*.

ISBN 1-880446-20-0